

El gran libro de HTML5, CSS3 y Javascript

Juan Diego Gauchat

marcombo
ediciones técnicas

Índice

El gran libro de HTML5, CSS3 y Javascript

Juan Diego Gauchat

marcombo
ediciones técnicas

Página de créditos

El gran libro de HTML5, CSS3 y Javascript

Primera edición en libro electrónico: Enero de 2012

Gran Via de les Corts Catalanes, 594
08007 Barcelona (España)
www.marcombo.com

ISBN edición en formato electrónico: 978-84-267-1782-5

ISBN edición en papel: 978-84-267-1770-2

Introducción

HTML5 no es una nueva versión del antiguo lenguaje de etiquetas, ni siquiera una mejora de esta ya antigua tecnología, sino un nuevo concepto para la construcción de sitios web y aplicaciones en una era que combina dispositivos móviles, computación en la nube y trabajos en red.

Todo comenzó mucho tiempo atrás con una simple versión de HTML propuesta para crear la estructura básica de páginas web, organizar su contenido y compartir información. El lenguaje y la web misma nacieron principalmente con la intención de comunicar información por medio de texto.

El limitado objetivo de HTML motivó a varias compañías a desarrollar nuevos lenguajes y programas para agregar características a la web nunca antes implementadas. Estos desarrollos iniciales crecieron hasta convertirse en populares y poderosos accesorios. Simples juegos y bromas animadas pronto se transformaron en sofisticadas aplicaciones, ofreciendo nuevas experiencias que cambiaron el concepto de la web para siempre.

De las opciones propuestas, Java y Flash fueron las más exitosas; ambas fueron masivamente adoptadas y ampliamente consideradas como el futuro de Internet. Sin embargo, tan pronto como el número de usuarios se incrementó e Internet pasó de ser una forma de conectar amantes de los ordenadores a un campo estratégico para los negocios y la interacción social, limitaciones presentes en estas dos tecnologías probaron ser una sentencia de muerte.

El mayor inconveniente de Java y Flash puede describirse como una falta de integración. Ambos fueron concebidos desde el principio como complementos (plug-ins), algo que se inserta dentro de una estructura pero que comparte con la misma solo espacio en la pantalla. No existía comunicación e integración alguna entre aplicaciones y documentos.

La falta de integración resultó ser crítica y preparó el camino para la evolución de un lenguaje que comparte espacio en el documento con HTML y no está afectado por las limitaciones de los *plug-ins*. Javascript, un lenguaje interpretado incluido en navegadores, claramente era la manera de mejorar la experiencia de los usuarios y proveer funcionalidad para la web. Sin embargo, después de algunos años de intentos fallidos para promoverlo y algunos malos usos, el mercado nunca lo adoptó plenamente y pronto su popularidad declinó. Los detractores tenían buenas razones para oponerse a su adopción. En ese momento, Javascript no era capaz de reemplazar la funcionalidad de Flash o Java. A pesar de ser evidente que ambos limitaban el alcance de las aplicaciones y aislaban el contenido web, populares funciones como la reproducción de video se estaban convirtiendo en una parte esencial de la web y solo eran efectivamente ofrecidas a través de estas tecnologías.

A pesar del suceso inicial, el uso de Java comenzó a declinar. La naturaleza compleja del lenguaje, su evolución lenta y la falta de integración disminuyeron su importancia hasta el punto en el que hoy día no es más usado en aplicaciones web de importancia. Sin Java, el mercado volcó su atención a Flash. Pero el hecho de que Flash comparte las mismas características básicas que su competidor en la web lo hace también susceptible de correr el mismo destino.

Mientras esta competencia silenciosa se llevaba a cabo, el software para acceder a la web continuaba evolucionando. Junto con nuevas funciones y técnicas rápidas de acceso a la red, los navegadores también mejoraron gradualmente sus intérpretes Javascript. Más potencia trajo más oportunidades y este lenguaje estaba listo para aprovecharlas.

En cierto punto durante este proceso, se hizo evidente para algunos desarrolladores que ni Java o Flash podrían proveer las herramientas que ellos necesitaban para crear las aplicaciones demandadas por un número creciente de usuarios. Estos desarrolladores, impulsados por las mejoras otorgadas por los navegadores, comenzaron a aplicar Javascript en sus aplicaciones de un modo nunca visto. La innovación y los increíbles resultados obtenidos llamaron la atención de más programadores. Pronto lo que fue llamado la “Web 2.0” nació y la percepción de Javascript en la comunidad de programadores cambió radicalmente.

Javascript era claramente el lenguaje que permitía a los desarrolladores innovar y hacer cosas que nadie había podido hacer antes en la web. En los últimos años, programadores y diseñadores web alrededor del mundo surgieron con los más increíbles trucos para superar las limitaciones de esta tecnología y sus iniciales deficiencias en portabilidad. Gracias a estas nuevas implementaciones, Javascript, HTML y CSS se convirtieron pronto en la más perfecta combinación para la necesaria evolución de la web.

HTML5 es, de hecho, una mejora de esta combinación, el pegamento que une todo. HTML5 propone estándares para cada aspecto de la web y también un propósito claro para cada una de las tecnologías involucradas. A partir de ahora, HTML provee los elementos estructurales, CSS se encuentra concentrado en cómo volver esa estructura utilizable y atractiva a la vista, y Javascript tiene todo el poder necesario para proveer dinamismo y construir aplicaciones web completamente funcionales.

Las barreras entre sitios webs y aplicaciones finalmente han desaparecido. Las tecnologías requeridas para el proceso de integración están listas. El futuro de la web es prometedor y la evolución y combinación de estas tres tecnologías (HTML, CSS y Javascript) en una poderosa especificación está volviendo a Internet la plataforma líder de desarrollo. HTML5 indica claramente el camino.

> **IMPORTANTE:** En este momento no todos los navegadores soportan HTML5 y la mayoría de sus funciones se encuentran actualmente en estado de desarrollo. Recomendamos leer los capítulos y ejecutar los códigos con las últimas versiones de Google Chrome y Firefox. Google Chrome ya implementa muchas de las características de HTML5

y además es una buena plataforma para pruebas. Por otro lado, Firefox es uno de los mejores navegadores para desarrolladores y también provee total soporte para HTML5.

Sea cual fuere el navegador elegido, siempre tenga en mente que un buen desarrollador instala y prueba sus códigos en cada programa disponible en el mercado. Ejecute los códigos provistos en este libro en cada uno de los navegadores disponibles.

Para descargar las últimas versiones, visite los siguientes enlaces:

- *www.google.com/chrome*
- *www.apple.com/safari/download*
- *www.mozilla.com*
- *windows.microsoft.com*
- *www.opera.com*

En la conclusión del libro exploramos diferentes alternativas para hacer sus sitios webs y aplicaciones accesibles desde viejos navegadores e incluso aquellos que aún no están preparados para HTML5.

Capítulo 1
Documentos HTML5

1.1 Componentes básicos

HTML5 provee básicamente tres características: estructura, estilo y funcionalidad. Nunca fue declarado oficialmente pero, incluso cuando algunas APIs (Interface de Programación de Aplicaciones) y la especificación de CSS3 por completo no son parte del mismo, HTML5 es considerado el producto de la combinación de HTML, CSS y Javascript. Estas tecnologías son altamente dependientes y actúan como una sola unidad organizada bajo la especificación de HTML5. HTML está a cargo de la estructura, CSS presenta esa estructura y su contenido en la pantalla y Javascript hace el resto que (como veremos más adelante) es extremadamente significativo.

Más allá de esta integración, la estructura sigue siendo parte esencial de un documento. La misma provee los elementos necesarios para ubicar contenido estático o dinámico, y es también una plataforma básica para aplicaciones. Con la variedad de dispositivos para acceder a Internet y la diversidad de interfaces disponibles para interactuar con la web, un aspecto básico como la estructura se vuelve parte vital del documento. Ahora la estructura debe proveer forma, organización y flexibilidad, y debe ser tan fuerte como los fundamentos de un edificio.

Para trabajar y crear sitios webs y aplicaciones con HTML5, necesitamos saber primero cómo esa estructura es construida. Crear fundamentos fuertes nos ayudará más adelante a aplicar el resto de los componentes para aprovechar completamente estas nuevas tecnologías.

Por lo tanto, empecemos por lo básico, paso a paso. En este primer capítulo aprenderá cómo construir una plantilla para futuros proyectos usando los nuevos elementos HTML introducidos en HTML5.

Hágalo usted mismo: Cree un archivo de texto vacío utilizando su editor de textos favorito para probar cada código presentado en este capítulo. Esto lo ayudará a recordar las nuevas etiquetas HTML y acostumbrarse a ellas.

Conceptos básicos: Un documento HTML es un archivo de texto. Si usted no posee ningún programa para desarrollo web, puede simplemente utilizar el Bloc de Notas de Windows o cualquier otro editor de textos. El archivo debe ser grabado con la extensión **`.html`** y el nombre que desee (por ejemplo, **`micodigo.html`**).

IMPORTANTE: Para acceder a información adicional y a los listados de ejemplo, visite nuestro sitio web *www.minkbooks.com*.

1.2 Estructura global

Los documentos HTML se encuentran estrictamente organizados. Cada parte del documento está diferenciada, declarada y determinada por etiquetas específicas. En esta parte del capítulo vamos a ver cómo construir la estructura global de un documento HTML y los nuevos elementos semánticos incorporados en HTML5.

<!DOCTYPE>

En primer lugar necesitamos indicar el tipo de documento que estamos creando. Esto en HTML5 es extremadamente sencillo:

```
<!DOCTYPE html>
```

Listado 1-1. *Usando el elemento <doctype>.*

IMPORTANTE: Esta línea debe ser la primera línea del archivo, sin espacios o líneas que la precedan. De esta forma, el modo estándar del navegador es activado y las incorporaciones de HTML5 son interpretadas siempre que sea posible, o ignoradas en caso contrario.

Hágalo usted mismo: Puede comenzar a copiar el código en su archivo de texto y agregar los próximos a medida que los vamos estudiando.

<html>

Luego de declarar el tipo de documento, debemos comenzar a construir la estructura HTML. Como siempre, la estructura tipo árbol de este lenguaje tiene su raíz en el elemento **<html>**. Este elemento envolverá al resto del código:

```
<!DOCTYPE html>
<html lang=”es”>
</html>
```

Listado 1-2. *Usando el elemento <html>.*

El atributo **lang** en la etiqueta de apertura **<html>** es el único atributo que necesitamos especificar en HTML5. Este atributo define el idioma humano del contenido del documento que estamos creando, en este caso **es** por español.

Conceptos básicos: HTML usa un lenguaje de etiquetas para construir páginas web. Estas etiquetas HTML son palabras clave y atributos rodeados de los signos mayor y menor (por ejemplo, **<html lang="es">**). En este caso, **html** es la palabra clave y **lang** es el atributo con el valor **es**. La mayoría de las etiquetas HTML se utilizan en pares, una etiqueta de apertura y una de cierre, y el contenido se declara entre ellas. En nuestro ejemplo, **<html lang="es">** indica el comienzo del código HTML y **</html>** indica el final. Compare las etiquetas de apertura y cierre y verá que la de cierre se distingue por una barra invertida antes de la palabra clave (por ejemplo, **</html>**). El resto de nuestro código será insertado entre estas dos etiquetas: **<html> ... </html>**.

IMPORTANTE: HTML5 es extremadamente flexible en cuanto a la estructura y a los elementos utilizados para construirla. El elemento **<html>** puede ser incluido sin ningún atributo o incluso ignorado completamente. Con el propósito de preservar compatibilidad (y por algunas razones extras que no vale la pena mencionar aquí) le recomendamos que siga algunas reglas básicas. En este libro vamos a enseñarle cómo construir documentos HTML de acuerdo a lo que nosotros consideramos prácticas recomendadas.

Para encontrar otros lenguajes para el atributo **lang** puede visitar el siguiente enlace: *www.w3schools.com/tags/ref_language_codes.asp.*

<head>

Continuemos construyendo nuestra plantilla. El código HTML insertado entre las etiquetas **<html>** tiene que ser dividido entre

dos secciones principales. Al igual que en versiones previas de HTML, la primera sección es la cabecera y la segunda el cuerpo. El siguiente paso, por lo tanto, será crear estas dos secciones en el código usando los elementos **<head>** y **<body>** ya conocidos.

El elemento **<head>** va primero, por supuesto, y al igual que el resto de los elementos estructurales tiene una etiqueta de apertura y una de cierre:

```
<!DOCTYPE html>
<html lang="es">
<head>

</head>

</html>
```

Listado 1-3. *Usando el elemento <head>.*

La etiqueta no cambió desde versiones anteriores y su propósito sigue siendo exactamente el mismo. Dentro de las etiquetas **<head>** definiremos el título de nuestra página web, declararemos el set de caracteres correspondiente, proveeremos información general acerca del documento e incorporaremos los archivos externos con estilos, códigos Javascript o incluso imágenes necesarias para generar la página en la pantalla.

Excepto por el título y algunos íconos, el resto de la información incorporada en el documento entre estas etiquetas es invisible para el usuario.

<body>

La siguiente gran sección que es parte principal de la organización de un documento HTML es el cuerpo. El cuerpo representa la parte visible de todo documento y es especificado entre etiquetas **<body>**. Estas etiquetas tampoco han cambiado en relación con versiones previas de HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>

</head>
<body>

</body>
</html>
```

Listado 1-4. *Usando el elemento <body>.*

Conceptos básicos: Hasta el momento tenemos un código simple pero con una estructura compleja. Esto es porque el código HTML no está formado por un conjunto de instrucciones secuenciales. HTML es un lenguaje de etiquetas, un listado de elementos que usualmente se utilizan en pares y que pueden ser anidados (totalmente contenidos uno dentro del otro). En la primera línea del código del Listado 1-4 tenemos una etiqueta simple con la definición del tipo de documento e inmediatamente después la etiqueta de apertura **<html lang="es">**. Esta etiqueta y la de cierre **</html>** al final del listado están indicando el comienzo del código HTML y su final. Entre las etiquetas **<html>** insertamos otras etiquetas especificando dos importantes partes de la estructura básica: **<head>** para la cabecera y **<body>** para el cuerpo del documento. Estas dos etiquetas también se utilizan en pares. Más adelante en este capítulo veremos que más etiquetas son insertadas entre estas últimas conformando una estructura de árbol con **<html>** como su raíz.

<meta>

Es momento de construir la cabecera del documento. Algunos cambios e innovaciones fueron incorporados dentro de la cabecera, y uno de ellos es la etiqueta que define el juego de caracteres a utilizar para mostrar el documento. Ésta es una etiqueta **<meta>** que especifica cómo el texto será presentado en pantalla:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">

</head>
<body>

</body>
</html>
```

Listado 1-5. *Usando el elemento <meta>.*

La innovación de este elemento en HTML5, como en la mayoría de los casos, es solo simplificación. La nueva etiqueta **<meta>** para la definición del tipo de caracteres es más corta y simple. Por supuesto, podemos cambiar el tipo **iso-8859-1** por el necesario para nuestros documentos y agregar otras etiquetas **<meta>** como **description** o **keywords** para definir otros aspectos de la página web, como es mostrado en el siguiente ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, Javascript">

</head>
<body>

</body>
</html>
```

Listado 1-6. *Agregando más elementos <meta>.*

> **Conceptos básicos:** Hay varios tipos de etiqueta **<meta>** que pueden ser incluidas para declarar información general sobre el documento, pero esta información no es mostrada en la ventana del navegador, es solo importante para motores de búsqueda y dispositivos que necesitan hacer una vista previa del documento u obtener un sumario de la información que contiene. Como comentamos anteriormente, aparte del título y algunos íconos, la mayoría de la información insertada entre las etiquetas **<head>** no es visible para los usuarios. En el código del Listado 1-6, el atributo **name** dentro de la etiqueta **<meta>** especifica su tipo y **content** declara su valor, pero ninguno de estos valores es mostrado en pantalla. Para aprender más sobre la etiqueta **<meta>**, visite nuestro sitio web y siga los enlaces proporcionados para este capítulo.

En HTML5 no es necesario cerrar etiquetas simples con una barra al final, pero recomendamos utilizarlas por razones de compatibilidad. El anterior código se podría escribir de la siguiente manera:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1" />
  <meta name="description" content="Ejemplo de HTML5" />
  <meta name="keywords" content="HTML5, CSS3, JavaScript" />

</head>
<body>

</body>
</html>
```

Listado 1-7. *Cierre de etiquetas simples.*

<title>

La etiqueta **<title>**, como siempre, simplemente especifica el título del documento, y no hay nada nuevo para comentar:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>

</head>
<body>

</body>
</html>
```

Listado 1-8. *Usando la etiqueta <title>.*

> **Conceptos básicos:** El texto entre las etiquetas **<title>** es el título del documento que estamos creando. Normalmente este texto es mostrado en la barra superior de la ventana del navegador.

<link>

Otro importante elemento que va dentro de la cabecera del documento es **<link>**. Este elemento es usado para incorporar estilos, códigos Javascript, imágenes o iconos desde archivos externos. Uno de los usos más comunes para **<link>** es la incorporación de archivos con estilos CSS:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel=”stylesheet” href=”misestilos.css”>
</head>
<body>

</body>
</html>
```

Listado 1-9. *Usando el elemento <link>.*

En HTML5 ya no se necesita especificar qué tipo de estilos estamos insertando, por lo que el atributo **type** fue eliminado. Solo necesitamos dos atributos para incorporar nuestro archivo de estilos: **rel** y **href**. El atributo **rel** significa “relación” y es acerca de la relación entre el documento y el archivo que estamos incorporando por medio de **href**. En este caso, el atributo **rel** tiene el valor **stylesheet** que le dice al navegador que el archivo **misestilos.css** es un archivo CSS con estilos requeridos para presentar la página en pantalla (en el próximo capítulo estudiaremos cómo utilizar estilos CSS).

> **Conceptos básicos:** Un archivo de estilos es un grupo de reglas de formato que ayudarán a cambiar la apariencia de nuestra página web (por ejemplo, el tamaño y color del texto). Sin estas reglas, el texto y cualquier otro elemento HTML sería mostrado en pantalla utilizando los estilos estándar provistos por el navegador. Los estilos son reglas simples que normalmente requieren solo unas pocas líneas de código y pueden ser declarados en el mismo documento. Como veremos más adelante, no es estrictamente necesario obtener esta información de archivos externos pero es una práctica recomendada. Cargar las reglas CSS desde un documento externo (otro archivo) nos permitirá organizar el documento principal, incrementar la velocidad de carga y aprovechar las nuevas características de HTML5.

Con esta última inserción podemos considerar finalizado nuestro trabajo en la cabecera. Ahora es tiempo de trabajar en el cuerpo, donde la magia ocurre.

1.3 Estructura del cuerpo

La estructura del cuerpo (el código entre las etiquetas **<body>**) generará la parte visible del documento. Este es el código que producirá nuestra página web.

HTML siempre ofreció diferentes formas de construir y organizar la información dentro del cuerpo de un documento. Uno de los primeros elementos provistos para este propósito fue **<table>**. Las tablas permitían a los diseñadores acomodar datos, texto, imágenes y herramientas dentro de filas y columnas de celdas, incluso sin que hayan sido concebidas para este propósito.

En los primeros días de la web, las tablas fueron una revolución, un gran paso hacia adelante con respecto a la visualización de los documentos y la experiencia ofrecida a los usuarios. Más adelante, gradualmente, otros elementos reemplazaron su función, permitiendo lograr lo mismo con menos código, facilitando de este modo la creación, permitiendo portabilidad y ayudando al mantenimiento de los sitios web.

El elemento **<div>** comenzó a dominar la escena. Con el surgimiento de webs más interactivas y la integración de HTML, CSS y Javascript, el uso de **<div>** se volvió una práctica común. Pero este elemento, así como **<table>**, no provee demasiada información acerca de las parte del cuerpo que está representando. Desde imágenes a menús, textos, enlaces, códigos, formularios, cualquier cosa puede ir entre las etiquetas de apertura y cierre de un elemento **<div>**. En otras palabras, la palabra clave **div** solo especifica una división en el cuerpo, como la celda de una tabla, pero no ofrece indicio alguno sobre qué clase de división es, cuál es su propósito o qué contiene.

Para los usuarios estas claves o indicios no son importantes, pero para los navegadores la correcta interpretación de qué hay dentro del documento que se está procesando puede ser crucial en muchos casos. Luego de la revolución de los dispositivos móviles y el surgimiento de diferentes formas en que la gente accede a la web, la identificación de cada parte del documento es una tarea que se ha vuelto más relevante que nunca.

Considerando todo lo expuesto, HTML5 incorpora nuevos elementos que ayudan a identificar cada sección del documento y organizar el cuerpo del mismo. En HTML5 las secciones más importantes son diferenciadas y la estructura principal ya no depende más de los elementos **<div>** o **<table>**.

Cómo usamos estos nuevos elementos depende de nosotros, pero las palabras clave otorgadas a cada uno de ellos nos dan ayudan a entender sus funciones. Normalmente una página o aplicación web está dividida entre varias áreas visuales para mejorar la experiencia del usuario y facilitar la interactividad. Las palabras claves que representan cada nuevo elemento de HTML5 están íntimamente relacionadas con estas áreas, como veremos pronto.

Organización

La Figura 1-1 representa un diseño común encontrado en la mayoría de los sitios webs estos días. Apesar del hecho de que cada diseñador crea sus propios diseños, en general podremos identificar las siguientes secciones en cada sitio web estudiado:



Figura 1-1. *Representación visual de un clásico diseño web.*

En la parte superior, descripto como **Cabecera**, se encuentra el espacio donde usualmente se ubica el logo, título, subtítulos y una corta descripción del sitio web o la página.

Inmediatamente debajo, podemos ver la **Barra de Navegación** en la cual casi todos los desarrolladores ofrecen un menú o lista de enlaces con el propósito de facilitar la navegación a través del sitio. Los usuarios son guiados desde esta barra hacia las diferentes páginas o documentos, normalmente pertenecientes al mismo sitio web.

El contenido más relevante de una página web se encuentra, en casi todo diseño, ubicado en su centro. Esta sección presenta información y enlaces valiosos. La mayoría de las veces es dividida en varias filas y columnas. En el ejemplo de la Figura 1-1 se utilizaron solo dos columnas: **Información Principal** y **Barra Lateral**, pero esta sección es extremadamente flexible y normalmente diseñadores la adaptan acorde a sus necesidades insertando más columnas, dividiendo cada columna entre bloques más pequeños o generando diferentes distribuciones y combinaciones. El contenido presentado en esta parte del diseño es usualmente de alta prioridad. En el diseño de ejemplo, **Información Principal** podría contener una lista de artículos, descripción de productos, entradas de un blog o cualquier otra información importante, y la **Barra Lateral** podría mostrar una lista de enlaces apuntando hacia cada uno se esos ítems. En un blog, por ejemplo, esta última columna ofrecerá una lista de enlaces apuntando a cada entrada del blog, información acerca del autor, etc…

En la base de un diseño web clásico siempre nos encontramos con una barra más que aquí llamamos **Institucional**. La nombramos de esta manera porque esta es el área en donde normalmente se muestra información acerca del sitio web, el autor o la empresa, además de algunos enlaces con respecto a reglas, términos y condiciones y toda información adicional que el desarrollador considere importante compartir. La barra **Institucional** es un complemento de la **Cabecera** y es parte de lo que se considera estos días la estructura esencial de una página web, como podemos apreciar en el siguiente ejemplo:



Figura 1-2. *Representación visual de un clásico diseño para blogs.*

La Figura 1-2 es una representación de un blog normal. En este ejemplo se puede claramente identificar cada parte del diseño considerado anteriormente.

1. **Cabecera**
2. **Barra de Navegación**
3. Sección de **Información Principal**
4. **Barra Lateral**
5. El pie o la barra **Institucional**

Esta simple representación de un blog nos puede ayudar a entender que cada sección definida en un sitio web tiene un propósito. A veces este propósito no es claro pero en esencia se encuentra siempre allí, ayudándonos a reconocer cualquiera de las secciones descriptas anteriormente en todo diseño.

HTML5 considera esta estructura básica y provee nuevos elementos para diferenciar y declarar cada una de sus partes. A partir de ahora podemos decir al navegador para qué es cada sección:

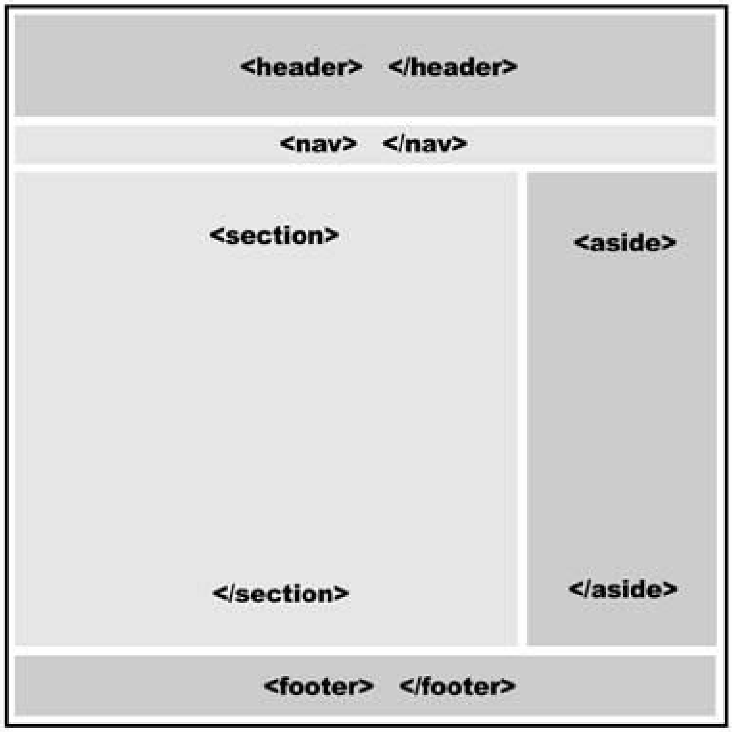


Figura 1-3. *Representación visual de un diseño utilizando elementos HTML5.*

La Figura 1-3 muestra el típico diseño presentado anteriormente, pero esta vez con los correspondientes elementos HTML5 para cada sección (incluyendo etiquetas de apertura y cierre).

<header>

Uno de los nuevos elementos incorporados en HTML5 es **<header>**. El elemento **<header>** no debe ser confundido con **<head>** usado antes para construir la cabecera del documento. Del mismo modo que **<head>**, la intención de **<header>** es proveer información introductoria (títulos, subtítulos, logos), pero difiere con respecto a **<head>** en su alcance. Mientras que el elemento **<head>** tiene el propósito de proveer información acerca de todo el documento, **<header>** es usado solo para el cuerpo o secciones específicas dentro del cuerpo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>

</body>
</html>
```

Listado 1-10. *Usando el elemento <header>.*

En el Listado 1-10, definimos el título de la página web utilizando el elemento **<header>**. Recuerde que esta cabecera no es la misma que la utilizada previamente para definir el título del documento. La inserción del elemento **<header>** representa el comienzo del cuerpo y por lo tanto de la parte visible del documento. De ahora en más será posible ver los resultados de

nuestro código en la ventana del navegador.

> **Hágalo usted mismo:** Si siguió las instrucciones desde el comienzo de este capítulo ya debería contar con un archivo de texto creado con todos los códigos estudiados hasta el momento y listo para ser probado. Si no es así, todo lo que debe hacer es copiar el código en el Listado 1-10 dentro de un archivo de texto vacío utilizando cualquier editor de texto (como el Bloc de Notas de Windows, por ejemplo) y grabar el archivo con el nombre de su agrado y la extensión **`.html`**. Para ver el código en funcionamiento, abra el archivo en un navegador compatible con HTML5 (puede hacerlo con un doble clic sobre el archivo en su explorador de archivos).

> **Conceptos básicos:** Entre las etiquetas **<header>** en el Listado 1-10 hay un elemento que probablemente no conoce. El elemento **<h1>** es un viejo elemento HTML usado para definir títulos. El número indica la importancia del título. El elemento **<h1>** es el más importante y **<h6>** el de menor importancia, por lo tanto **<h1>** será utilizado para mostrar el título principal y los demás para subtítulos o subtítulos internos. Más adelante veremos cómo estos elementos trabajan en HTML5.

<nav>

Siguiendo con nuestro ejemplo, la siguiente sección es la **Barra de Navegación**. Esta barra es generada en HTML5 con el elemento **<nav>**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>

</body>
</html>
```

Listado 1-11. *Usando el elemento <nav>.*

Como se puede apreciar en el Listado 1-11, el elemento **<nav>** se encuentra dentro de las etiquetas **<body>** pero es ubicado después de la etiqueta de cierre de la cabecera (**</header>**), no dentro de las etiquetas **<header>**. Esto es porque **<nav>** no es parte de la cabecera sino una nueva sección.

Anteriormente dijimos que la estructura y el orden que elegimos para colocar los elementos HTML5 dependen de nosotros. Esto significa que HTML5 es versátil y solo nos otorga los parámetros y elementos básicos con los que trabajar, pero cómo usarlos será exclusivamente decisión nuestra. Un ejemplo de esta versatilidad es que el elemento **<nav>** podría ser insertado dentro del elemento **<header>** o en cualquier otra parte del cuerpo. Sin embargo, siempre se debe considerar que estas etiquetas fueron creadas para brindar información a los navegadores y ayudar a cada nuevo programa y dispositivo en el mercado a identificar las partes más relevantes del documento. Para conservar nuestro código portable y comprensible, recomendamos como buena práctica seguir lo que marcan los estándares y mantener todo tan claro como sea posible. El elemento **<nav>** fue creado para ofrecer ayuda para la navegación, como en menús principales o grandes bloques de enlaces, y debería ser utilizado de esa manera.

> **Conceptos básicos:** En el ejemplo del Listado 1-11 generamos las opciones del menú para nuestra página web. Entre las etiquetas **<nav>** hay dos elementos que son utilizados para crear una lista. El propósito del elemento **<ul>** es definir la lista. Anidado entre las etiquetas **<ul>** encontramos varias etiquetas **<li>** con diferentes textos representando

las opciones del menú. Las etiquetas **<li>**, como probablemente ya se ha dado cuenta, son usadas para definir cada ítem de la lista. El propósito de este libro no es enseñarle conceptos básicos sobre HTML, si necesita más información acerca de elementos regulares de este lenguaje visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

<section>

Siguiendo nuestro diseño estándar nos encontramos con las columnas que en la Figura 1-1 llamamos **Información Principal** y **Barra Lateral**. Como explicamos anteriormente, la columna **Información Principal** contiene la información más relevante del documento y puede ser encontrada en diferentes formas (por ejemplo, dividida en varios bloques o columnas). Debido a que el propósito de estas columnas es más general, el elemento en HTML5 que especifica estas secciones se llama simplemente **<section>**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section>

  </section>

</body>
</html>
```

Listado 1-12. *Usando el elemento <section>.*

Al igual que la **Barra de Navegación**, la columna **Información Principal** es una sección aparte. Por este motivo, la sección para **Información Principal** va debajo de la etiqueta de cierre **</nav>**.

Hágalo usted mismo: Compare el último código en el Listado 1-12 con el diseño de la Figura 1-3 para comprender cómo las etiquetas son ubicadas en el código y qué sección cada una de ellas genera en la representación visual de la página web.

IMPORTANTE: Las etiquetas que representan cada sección del documento están localizadas en el código en forma de lista, unas sobre otras, pero en el sitio web algunas de estas secciones se ubicarán lado a lado (las columnas **Información Principal** y **Barra Lateral** son un claro ejemplo). En HTML5, la responsabilidad por la representación de los elementos en la pantalla fue delegada a CSS. El diseño será logrado asignando estilos CSS a cada elemento HTML. Estudiaremos CSS en el próximo capítulo.

<aside>

En un típico diseño web (Figura 1-1) la columna llamada **Barra Lateral** se ubica al lado de la columna **Información Principal**. Esta es una columna o sección que normalmente contiene datos relacionados con la información principal pero que no son relevantes o igual de importantes.

En el diseño de un blog, por ejemplo, la **Barra Lateral** contendrá una lista de enlaces. En el ejemplo de la Figura 1-2, los enlaces apuntan a cada una de las entradas del blog y ofrecen información adicional sobre el autor (número 4). La información dentro de esta barra está relacionada con la información principal pero no es relevante por sí misma. Siguiendo el mismo ejemplo podemos decir que las entradas del blog son relevantes pero los enlaces y las pequeñas reseñas sobre esas entradas son solo una ayuda para la navegación pero no lo que al lector realmente le interesa.

En HTML5 podemos diferenciar esta clase secundaria de información utilizando el elemento **`<aside>`**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section>

   </section>
   <aside>
     <blockquote>Mensaje número uno</blockquote>
     <blockquote>Mensaje número dos</blockquote>
   </aside>

</body>
</html>
```

***Listado 1-13.** Usando el elemento <aside>.*

El elemento **`<aside>`** podría estar ubicado del lado derecho o izquierdo de nuestra página de ejemplo, la etiqueta no tiene una posición predefinida. El elemento **`<aside>`** solo describe la información que contiene, no el lugar dentro de la estructura. Este elemento puede estar ubicado en cualquier parte del diseño y ser usado siempre y cuando su contenido no sea considerado como el contenido principal del documento. Por ejemplo, podemos usar **`<aside>`** dentro del elemento **`<section>`** o incluso insertado entre la información relevante, como en el caso de una cita.

<footer>

Para finalizar la construcción de la plantilla o estructura elemental de nuestro documento HTML5, solo necesitamos un elemento más. Ya contamos con la cabecera del cuerpo, secciones con ayuda para la navegación, información importante y hasta una barra lateral con datos adicionales, por lo tanto lo único que nos queda por hacer es cerrar nuestro diseño para otorgarle un final al cuerpo del documento. HTML5 provee un elemento específico para este propósito llamado **`<footer>`**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
```

```
    <link rel="stylesheet" href="misestilos.css">
  </head>
  <body>
    <header>
      <h1>Este es el título principal del sitio web</h1>
    </header>
    <nav>
      <ul>
        <li>principal</li>
        <li>fotos</li>
        <li>videos</li>
        <li>contacto</li>
      </ul>
    </nav>
    <section>

    </section>
    <aside>
      <blockquote>Mensaje número uno</blockquote>
      <blockquote>Mensaje número dos</blockquote>
    </aside>
    <footer>
      Derechos Reservados &copy; 2010-2011
    </footer>

  </body>
  </html>
```

Listado 1-14. *Usando el elemento <footer>.*

En el típico diseño de una página web (Figura 1-1) la sección llamada **Institucional** será definida por etiquetas **`<footer>`**. Esto es debido a que la barra representa el final (o pie) del documento y esta parte de la página web es normalmente usada para compartir información general sobre el autor o la organización detrás del proyecto.

Generalmente, el elemento **`<footer>`** representará el final del cuerpo de nuestro documento y tendrá el propósito descripto anteriormente. Sin embargo, **`<footer>`** puede ser usado múltiples veces dentro del cuerpo para representar también el final de diferentes secciones (del mismo modo que la etiqueta **`<header>`**). Estudiaremos esta última característica más adelante.

1.4 Dentro del cuerpo

El cuerpo de nuestro documento está listo. La estructura básica de nuestro sitio web fue finalizada, pero aún tenemos que trabajar en el contenido. Los elementos HTML5 estudiados hasta el momento nos ayudan a identificar cada sección del diseño y asignar un propósito intrínseco a cada una de ellas, pero lo que es realmente importante para nuestro sitio web se encuentra en el interior de estas secciones.

La mayoría de los elementos ya estudiados fueron creados para construir una estructura para el documento HTML que pueda ser identificada y reconocida por los navegadores y nuevos dispositivos. Aprendimos acerca de la etiqueta **<body>** usada para declarar el cuerpo o parte visible del documento, la etiqueta **<header>** con la que agrupamos información importante para el cuerpo, la etiqueta **<nav>** que provee ayuda para la navegación del sitio web, la etiqueta **<section>** necesaria para contener la información más relevante, y también **<aside>** y **<footer>** para ofrecer información adicional de cada sección y del documento mismo. Pero ninguno de estos elementos declara algo acerca del contenido. Todos tienen un específico propósito estructural.

Más profundo nos introducimos dentro del documento más cerca nos encontramos de la definición del contenido. Esta información estará compuesta por diferentes elementos visuales como títulos, textos, imágenes, videos y aplicaciones interactivas, entre otros. Necesitamos poder diferenciar estos elementos y establecer una relación entre ellos dentro de la estructura.

<article>

El diseño considerado anteriormente (Figura 1-1) es el más común y representa una estructura esencial para los sitios web estos días, pero es además ejemplo de cómo el contenido clave es mostrado en pantalla. Del mismo modo que los blogs están divididos en entradas, sitios web normalmente presentan información relevante dividida en partes que comparten similares características. El elemento **<article>** nos permite identificar cada una de estas partes:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section>
    <article>
      Este es el texto de mi primer mensaje
    </article>
    <article>
      Este es el texto de mi segundo mensaje
    </article>
  </section>
  <aside>
    <blockquote>Mensaje número uno</blockquote>
    <blockquote>Mensaje número dos</blockquote>
  </aside>
  <footer>
    Derechos Reservados &copy; 2010-2011
```

```
  </footer>
</body>
</html>
```

Listado 1-15. *Usando el elemento <article>.*

Como puede observarse en el código del Listado 1-15, las etiquetas **<article>** se encuentran ubicadas dentro del elemento **<section>**. Las etiquetas **<article>** en nuestro ejemplo pertenecen a esta sección, son sus hijos, del mismo modo que cada elemento dentro de las etiquetas **<body>** es hijo del cuerpo. Y al igual que cada elemento hijo del cuerpo, las etiquetas **<article>** son ubicadas una sobre otra, como es mostrado en la Figura 1-4.

Conceptos básicos: Como dijimos anteriormente, la estructura de un documento HTML puede ser descripta como un árbol, con el elemento **<html>** como su raíz. Otra forma de describir la relación entre elementos es nombrarlos como padres, hijos y hermanos, de acuerdo a la posición que ocupan dentro de esa misma estructura. Por ejemplo, en un típico documento HTML el elemento **<body>** es hijo del elemento **<html>** y hermano del elemento **<head>**. Ambos, **<body>** y **<head>**, tienen al elemento **<html>** como su padre.

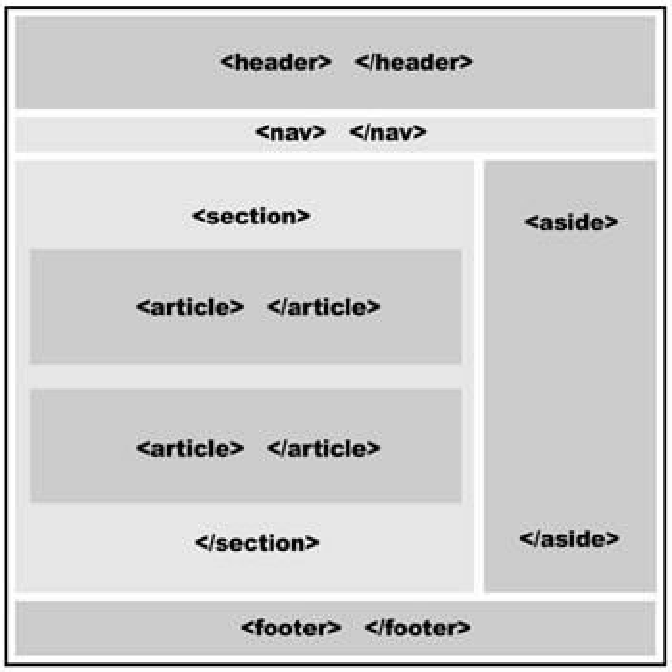


Figura 1-4. *Representación visual de las etiquetas <article> que fueron incluidas para contener información relevante de la página web.*

El elemento **<article>** no está limitado por su nombre (no se limita, por ejemplo, a artículos de noticias). Este elemento fue creado con la intención de contener unidades independientes de contenido, por lo que puede incluir mensajes de foros, artículos de una revista digital, entradas de blog, comentarios de usuarios, etc… Lo que hace es agrupar porciones de información que están relacionadas entre sí independientemente de su naturaleza.

Como una parte independiente del documento, el contenido de cada elemento **<article>** tendrá su propia estructura. Para definir esta estructura, podemos aprovechar la versatilidad de los elementos **<header>** y **<footer>** estudiados anteriormente. Estos elementos son portables y pueden ser usados no solo para definir los límites del cuerpo sino también en cualquier sección de nuestro documento:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
```

```
    <header>
      <h1>Este es el título principal del sitio web</h1>
    </header>
    <nav>
      <ul>
        <li>principal</li>
        <li>fotos</li>
        <li>videos</li>
        <li>contacto</li>
      </ul>
    </nav>
    <section>
      <article>
        <header>
          <h1>Título del mensaje uno</h1>
        </header>
        Este es el texto de mi primer mensaje
        <footer>
          <p>comentarios (0)</p>
        </footer>
      </article>
      <article>
        <header>
          <h1>Titulo del mensaje dos</h1>
        </header>
        Este es el texto de mi segundo mensaje
        <footer>
          <p>comentarios (0)</p>
        </footer>
      </article>
    </section>
    <aside>
      <blockquote>Mensaje número uno</blockquote>
      <blockquote>Mensaje número dos</blockquote>
    </aside>
    <footer>
      Derechos Reservados &copy; 2010-2011
    </footer>
  </body>
  </html>
```

Listado 1-16. *Construyendo la estructura de <article>.*

Los dos mensajes insertados en el código del Listado 1-16 fueron construidos con el elemento **<article>** y tienen una estructura específica. En la parte superior de esta estructura incluimos las etiquetas **<header>** conteniendo el título definido con el elemento **<h1>**, debajo se encuentra el contenido mismo del mensaje y sobre el final, luego del texto, vienen las etiquetas **<footer>** especificando la cantidad de comentarios recibidos.

<hgroup>

Dentro de cada elemento **<header>**, en la parte superior del cuerpo o al comienzo de cada **<article>**, incorporamos elementos **<h1>** para declarar un título. Básicamente, las etiquetas **<h1>** son todo lo que necesitamos para crear una línea de cabecera para cada parte del documento, pero es normal que necesitemos también agregar subtítulos o más información que especifique de qué se trata la página web o una sección en particular. De hecho, el elemento **<header>** fue creado para contener también otros elementos como tablas de contenido, formularios de búsqueda o textos cortos y logos.

Para construir este tipo de cabeceras, podemos aprovechar el resto de las etiquetas H, como **<h1>, <h2>**, **<h3>**, **<h4>**, **<h5>** y **<h6>**, pero siempre considerando que por propósitos de procesamiento interno, y para evitar generar múltiples secciones durante la interpretación del documento por parte del navegador, estas etiquetas deben ser agrupadas juntas. Por esta razón, HTML5 provee el elemento **<hgroup>**:

```
  <!DOCTYPE html>
  <html lang="es">
```

```
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section>
    <article>
      <header>
        <hgroup>
          <h1>Título del mensaje uno</h1>
          <h2>Subtítulo del mensaje uno</h2>
        </hgroup>
        <p>publicado 10-12-2011</p>
      </header>
      Este es el texto de mi primer mensaje
      <footer>
        <p>comentarios (0)</p>
      </footer>
    </article>
    <article>
      <header>
        <hgroup>
          <h1>Título del mensaje dos</h1>
          <h2>Subtítulo del mensaje dos</h2>
        </hgroup>
        <p>publicado 15-12-2011</p>
      </header>
      Este es el texto de mi segundo mensaje
      <footer>
        <p>comentarios (0)</p>
      </footer>
    </article>
  </section>
  <aside>
    <blockquote>Mensaje número uno</blockquote>
    <blockquote>Mensaje número dos</blockquote>
  </aside>
  <footer>
    Derechos Reservados &copy; 2010-2011
  </footer>
</body>
</html>
```

Listado 1-17. *Usando el elemento <hgroup>.*

Las etiquetas H deben conservar su jerarquía, lo que significa que debemos primero declarar la etiqueta **<h1>**, luego usar **<h2>** para subtítulos y así sucesivamente. Sin embargo, a diferencia de anteriores versiones de HTML, HTML5 nos deja reusar las etiquetas H y construir esta jerarquía una y otra vez en cada sección del documento. En el ejemplo del Listado 1-17, agregamos un subtítulo y datos adicionales a cada mensaje. Los títulos y subtítulos fueron agrupados juntos utilizando **<hgroup>**, recreando de este modo la jerarquía **<h1>** y **<h2>** en cada elemento **<article>**.

IMPORTANTE: El elemento **<hgroup>** es necesario cuando tenemos un título y subtítulo o más etiquetas H juntas en la misma cabecera. Este elemento puede contener solo etiquetas H y esta fue la razón por la que en nuestro ejemplo

dejamos los datos adicionales afuera. Si solo dispone de una etiqueta **<h1>** o la etiqueta **<h1>** junto con datos adicionales, no tiene que agrupar estos elementos juntos. Por ejemplo, en la cabecera del cuerpo (**<header>**) no usamos este elemento porque solo tenemos una etiqueta H en su interior. Siempre recuerde que **<hgroup>** fue creado solo con la intención de agrupar etiquetas H, exactamente como su nombre lo indica.

Navegadores y programas que ejecutan y presentan en la pantalla sitios webs leen el código HTML y crean su propia estructura interna para interpretar y procesar cada elemento. Esta estructura interna está dividida en secciones que no tienen nada que ver con las divisiones en el diseño o el elemento **<section>**. Estas son secciones conceptuales generadas durante la interpretación del código. El elemento **<header>** no crea una de estas secciones por sí mismo, lo que significa que los elementos dentro de **<header>** representarán diferentes niveles e internamente pueden generar diferentes secciones. El elemento **<hgroup>** fue creado con el propósito de agrupar las etiquetas H y evitar interpretaciones incorrectas por parte de los navegadores.

Conceptos básicos: lo que llamamos "información adicional" dentro de la cabecera en nuestra descripción previa es conocido como Metadata. Metadata es un conjunto de datos que describen y proveen información acerca de otro grupo de datos. En nuestro ejemplo, Metadata es la fecha en la cual cada mensaje fue publicado.

<figure> y <figcaption>

La etiqueta **<figure>** fue creada para ayudarnos a ser aún más específicos a la hora de declarar el contenido del documento. Antes de que este elemento sea introducido, no podíamos identificar el contenido que era parte de la información pero a la vez independiente, como ilustraciones, fotos, videos, etc… Normalmente estos elementos son parte del contenido relevante pero pueden ser extraídos o movidos a otra parte sin afectar o interrumpir el flujo del documento. Cuando nos encontramos con esta clase de información, las etiquetas **<figure>** pueden ser usadas para identificarla:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <header>
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav>
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section>
    <article>
      <header>
        <hgroup>
          <h1>Título del mensaje uno</h1>
          <h2>Subtítulo del mensaje uno</h2>
        </hgroup>
        <p>publicado 10-12-2011</p>
      </header>
      Este es el texto de mi primer mensaje
      <figure>
        <img src="http://minkbooks.com/content/myimage.jpg">
        <figcaption>
           Esta es la imagen del primer mensaje
         </figcaption>
      </figure>
      <footer>
        <p>comentarios (0)</p>
```

```
        </footer>
      </article>
      <article>
        <header>
          <hgroup>
            <h1>Título del mensaje dos</h1>
            <h2>Subtítulo del mensaje dos</h2>
          </hgroup>
          <p>publicado 15-12-2011</p>
        </header>
        Este es el texto de mi segundo mensaje
        <footer>
          <p>comentarios (0)</p>
        </footer>
      </article>
    </section>
    <aside>
      <blockquote>Mensaje número uno</blockquote>
      <blockquote>Mensaje número dos</blockquote>
    </aside>
    <footer>
      Derechos Reservados &copy; 2010-2011
    </footer>
  </body>
  </html>
```

Listado 1-18. *Usando los elementos <figure> y <figcaption>.*

En el Listado 1-18, en el primer mensaje, luego del texto insertamos una imagen (**<img src="http://minkbooks.com/content/myimage.jpg">**). Esta es una práctica común, a menudo el texto es enriquecido con imágenes o videos. Las etiquetas **<figure>** nos permiten envolver estos complementos visuales y diferenciarlos así de la información más relevante.

También en el Listado 1-18 se puede observar un elemento extra dentro de **<figure>**. Normalmente, unidades de información como imágenes o videos son descriptas con un corto texto debajo. HTML5 provee un elemento para ubicar e identificar estos títulos descriptivos. Las etiquetas **<figcaption>** encierran el texto relacionado con **<figure>** y establecen una relación entre ambos elementos y su contenido.

1.5 Nuevos y viejos elementos

HTML5 fue desarrollado con la intención de simplificar, especificar y organizar el código. Para lograr este propósito, nuevas etiquetas y atributos fueron agregados y HTML fue completamente integrado a CSS y Javascript. Estas incorporaciones y mejoras de versiones previas están relacionadas no solo con nuevos elementos sino también con cómo usamos los ya existentes.

<mark>

La etiqueta **<mark>** fue agregada para resaltar parte de un texto que originalmente no era considerado importante pero ahora es relevante acorde con las acciones del usuario. El ejemplo que más se ajusta a este caso es un resultado de búsqueda. El elemento **<mark>** resaltará la parte del texto que concuerda con el texto buscado:

```
<span>Mi <mark>coche</mark> es rojo</span>
```

Listado 1-19. *Uso del elemento <mark> para resaltar la palabra “coche”.*

Si un usuario realiza una búsqueda de la palabra “coche”, por ejemplo, los resultados podrían ser mostrados con el código del Listado 1-19. La frase del ejemplo representa los resultados de la búsqueda y las etiquetas **<mark>** en el medio encierran lo que era el texto buscado (la palabra “coche”). En algunos navegadores, esta palabra será resaltada con un fondo amarillo por defecto, pero siempre podemos sobrescribir estos estilos con los nuestros utilizando CSS, como veremos en próximos capítulos.

En el pasado, normalmente obteníamos similares resultados usando el elemento **<b>**. El agregado de **<mark>** tiene el objetivo de cambiar el significado y otorgar un nuevo propósito para éstos y otros elementos relacionados:

- **<em>** es para indicar énfasis (reemplazando la etiqueta **<i>** que utilizábamos anteriormente).
- **<strong>** es para indicar importancia.
- **<mark>** es para resaltar texto que es relevante de acuerdo con las circunstancias.
- **<b>** debería ser usado solo cuando no hay otro elemento más apropiado para la situación.

<small>

La nueva especificidad de HTML es también evidente en elementos como **<small>**. Previamente este elemento era utilizado con la intención de presentar cualquier texto con letra pequeña. La palabra clave referenciaba el tamaño del texto, independientemente de su significado. En HTML5, el nuevo propósito de **<small>** es presentar la llamada letra pequeña, como impresiones legales, descargos, etc…

```
<small>Derechos Reservados &copy; 2011 MinkBooks</small>
```

Listado 1-20. *Inclusión de información legal con el elemento <small>.*

<cite>

Otro elemento que ha cambiado su naturaleza para volverse más específico es **<cite>**. Ahora las etiquetas **<cite>** encierran el título de un trabajo, como un libro, una película, una canción, etc…

```
<span>Amo la película <cite>Tentaciones</cite></span>
```

Listado 1-21. *Citando una película con el elemento <cite>.*

<address>

El elemento **<address>** es un viejo elemento convertido en un elemento estructural. No necesitamos usarlo previamente para construir nuestra plantilla, sin embargo podría ubicarse perfectamente en algunas situaciones en las que debemos presentar información de contacto relacionada con el contenido del elemento **<article>** o el cuerpo completo.

Este elemento debería ser incluido dentro de **<footer>**, como en el siguiente ejemplo:

```
<article>
  <header>
    <h1>Título del mensaje </h1>
  </header>
  Este es el texto del mensaje
  <footer>
    <address>
      <a href="http://www.jdgauchat.com">JD Gauchat</a>
    </address>
  </footer>
</article>
```

Listado 1-22. *Agregando información de contacto a un <article>.*

<time>

En cada **<article>** de nuestra última plantilla (Listado 1-18), incluimos la fecha indicando cuándo el mensaje fue publicado. Para esto usamos un simple elemento **<p>** dentro de la cabecera (**<header>**) del mensaje, pero existe un elemento en HTML5 específico para este propósito. El elemento **<time>** nos permite declarar un texto comprensible para humanos y navegadores que representa fecha y hora:

```
<article>
  <header>
    <h1>Título del mensaje dos</h1>
    <time datetime="2011-10-12" pubdate>publicado 12-10-2011</time>
  </header>
  Este es el texto del mensaje
</article>
```

Listado 1-23. *Fecha y hora usando el elemento <time>.*

En el Listado 1-23, el elemento **<p>** usado en ejemplos previos fue reemplazado por el nuevo elemento **<time>** para mostrar la fecha en la que el mensaje fue publicado. El atributo **datetime** tiene el valor que representa la fecha comprensible para el navegador (timestamp). El formato de este valor deberá seguir un patrón similar al del siguiente ejemplo: **2011-10-12T12:10:45**. También incluimos el atributo **pubdate**, el cual solo es agregado para indicar que el valor del atributo **datetime** representa la fecha de publicación.

1.6 Referencia rápida

En la especificación HTML5, HTML está a cargo de la estructura del documento y provee un grupo completo de nuevos elementos para este propósito. La especificación también incluye algunos elementos con la única tarea de proveer estilos. Esta es una lista de los que consideramos más relevantes:

IMPORTANTE: Para una completa referencia de los elementos HTML incluidos en la especificación, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

<header> Este elemento presenta información introductoria y puede ser aplicado en diferentes secciones del documento. Tiene el propósito de contener la cabecera de una sección pero también puede ser utilizado para agrupar índices, formularios de búsqueda, logos, etc…

<nav> Este elemento indica una sección de enlaces con propósitos de navegación, como menús o índices. No todos los enlaces dentro de una página web tienen que estar dentro de un elemento **`<nav>`**, solo aquellos que forman partes de bloques de navegación.

<section> Este elemento representa una sección general del documento. Es usualmente utilizado para construir varios bloques de contenido (por ejemplo, columnas) con el propósito de ordenar el contenido que comparte una característica específica, como capítulos o páginas de un libro, grupo de noticias, artículos, etc…

<aside> Este elemento representa contenido que está relacionado con el contenido principal pero no es parte del mismo. Ejemplos pueden ser citas, información en barras laterales, publicidad, etc…

<footer> Este elemento representa información adicional sobre su elemento padre. Por ejemplo, un elemento **`<footer>`** insertado al final del cuerpo proveerá información adicional sobre el cuerpo del documento, como el pie normal de una página web. Puede ser usado no solo para el cuerpo sino también para diferentes secciones dentro del cuerpo, otorgando información adicional sobre estas secciones específicas.

<article> Este elemento representa una porción independiente de información relevante (por ejemplo, cada artículo de un periódico o cada entrada de un blog). El elemento **`<article>`** puede ser anidado y usado para mostrar una lista dentro de otra lista de ítems relacionados, como comentarios de usuarios en entradas de blogs, por ejemplo.

<hgroup> Este elemento es usado para agrupar elementos H cuando la cabecera tiene múltiples niveles (por ejemplo, una cabecera con título y subtítulo).

<figure> Este elemento representa una porción independiente de contenido (por ejemplo, imágenes, diagramas o videos) que son referenciadas desde el contenido principal. Esta es información que puede ser removida sin afectar el fluido del resto del contenido.

<figcaption> Este elemento es utilizado para mostrar una leyenda o pequeño texto relacionado con el contenido de un elemento **`<figure>`**, como la descripción de una imagen.

<mark> Este elemento resalta un texto que tiene relevancia en una situación en particular o que ha sido mostrado en respuesta de la actividad del usuario.

<small> Este elemento representa contenido al margen, como letra pequeña (por ejemplo, descargos, restricciones legales, declaración de derechos, etc…).

<cite> Este elemento es usado para mostrar el título de un trabajo (libro, película, poema, etc…).

<address> Este elemento encierra información de contacto para un elemento **`<article>`** o el documento completo. Es recomendable que sea insertado dentro de un elemento **`<footer>`**.

<time> Este elemento se utiliza para mostrar fecha y hora en formatos comprensibles por los usuarios y el navegador. El valor para los usuarios es ubicado entre las etiquetas mientras que el específico para programas y navegadores es incluido como el valor del atributo **`datetime`**. Un segundo atributo optativo llamado **`pubdate`** es usado para indicar que el valor de **`datetime`** es la fecha de publicación.

Capítulo 2
Estilos CSS y modelos de caja

2.1 CSS y HTML

Como aclaramos anteriormente, la nueva especificación de HTML (HTML5) no describe solo los nuevos elementos HTML o el lenguaje mismo. La web demanda diseño y funcionalidad, no solo organización estructural o definición de secciones. En este nuevo paradigma, HTML se presenta junto con CSS y Javascript como un único instrumento integrado.

La función de cada tecnología ya ha sido explicada en capítulos previos, así como los nuevos elementos HTML responsables de la estructura del documento. Ahora es momento de analizar CSS, su relevancia dentro de esta unión estratégica y su influencia sobre la presentación de documentos HTML.

Oficialmente CSS nada tiene que ver con HTML5. CSS no es parte de la especificación y nunca lo fue. Este lenguaje es, de hecho, un complemento desarrollado para superar las limitaciones y reducir la complejidad de HTML. Al comienzo, atributos dentro de las etiquetas HTML proveían estilos esenciales para cada elemento, pero a medida que el lenguaje evolucionó, la escritura de códigos se volvió más compleja y HTML por sí mismo no pudo más satisfacer las demandas de diseñadores. En consecuencia, CSS pronto fue adoptado como la forma de separar la estructura de la presentación. Desde entonces, CSS ha crecido y ganado importancia, pero siempre desarrollado en paralelo, enfocado en las necesidades de los diseñadores y apartado del proceso de evolución de HTML.

La versión 3 de CSS sigue el mismo camino, pero esta vez con un mayor compromiso. La especificación de HTML5 fue desarrollada considerando CSS a cargo del diseño. Debido a esta consideración, la integración entre HTML y CSS es ahora vital para el desarrollo web y esta es la razón por la que cada vez que mencionamos HTML5 también estamos haciendo referencia a CSS3, aunque oficialmente se trate de dos tecnologías completamente separadas.

En este momento las nuevas características incorporadas en CSS3 están siendo implementadas e incluidas junto al resto de la especificación en navegadores compatibles con HTML5. En este capítulo, vamos a estudiar conceptos básicos de CSS y las nuevas técnicas de CSS3 ya disponibles para presentación y estructuración. También aprenderemos cómo utilizar los nuevos selectores y pseudo clases que hacen más fácil la selección e identificación de elementos HTML.

Conceptos básicos: CSS es un lenguaje que trabaja junto con HTML para proveer estilos visuales a los elementos del documento, como tamaño, color, fondo, bordes, etc…

IMPORTANTE: En este momento las nuevas incorporaciones de CSS3 están siendo implementadas en las últimas versiones de los navegadores más populares, pero algunas de ellas se encuentran aún en estado experimental. Por esta razón, estos nuevos estilos deberán ser precedidos por prefijos tales como **`-moz-`** o **`-webkit-`** para ser efectivamente interpretados. Analizaremos este importante asunto más adelante.

2.2 Estilos y estructura

A pesar de que cada navegador garantiza estilos por defecto para cada uno de los elementos HTML, estos estilos no necesariamente satisfacen los requerimientos de cada diseñador. Normalmente se encuentran muy distanciados de lo que queremos para nuestros sitios webs. Diseñadores y desarrolladores a menudo deben aplicar sus propios estilos para obtener la organización y el efecto visual que realmente desean.

> **IMPORTANTE:** En esta parte del capítulo vamos a revisar estilos CSS y explicar algunas técnicas básicas para definir la estructura de un documento. Si usted ya se encuentra familiarizado con estos conceptos, siéntase libre de obviar las partes que ya conoce.

Elementos block

Con respecto a la estructura, básicamente cada navegador ordena los elementos por defecto de acuerdo a su tipo: *block* (bloque) o *inline* (en línea). Esta clasificación está asociada con la forma en que los elementos son mostrados en pantalla.

- **Elementos *block*** son posicionados uno sobre otro hacia abajo en la página.
- **Elementos *inline*** son posicionados lado a lado, uno al lado del otro en la misma línea, sin ningún salto de línea a menos que ya no haya más espacio horizontal para ubicarlos.

Casi todos los elementos estructurales en nuestros documentos serán tratados por los navegadores como elementos *block* por defecto. Esto significa que cada elemento HTML que representa una parte de la organización visual (por ejemplo, **<section>**, **<nav>**, **<header>**, **<footer>**, **<div>**) será posicionado debajo del anterior.

En el Capítulo 1 creamos un documento HTML con la intención de reproducir un sitio web tradicional. El diseño incluyó barras horizontales y dos columnas en el medio. Debido a la forma en que los navegadores muestran estos elementos por defecto, el resultado en la pantalla está muy lejos de nuestras expectativas. Tan pronto como el archivo HTML con el código del Listado 1-18, Capítulo 1, es abierto en el navegador, la posición errónea en la pantalla de las dos columnas definidas por los elementos **<section>** y **<aside>** es claramente visible. Una columna está debajo de la otra en lugar de estar a su lado, como correspondería. Cada bloque (*block*) es mostrado por defecto tan ancho como sea posible, tan alto como la información que contiene y uno sobre otro, como se muestra en la Figura 2-1.

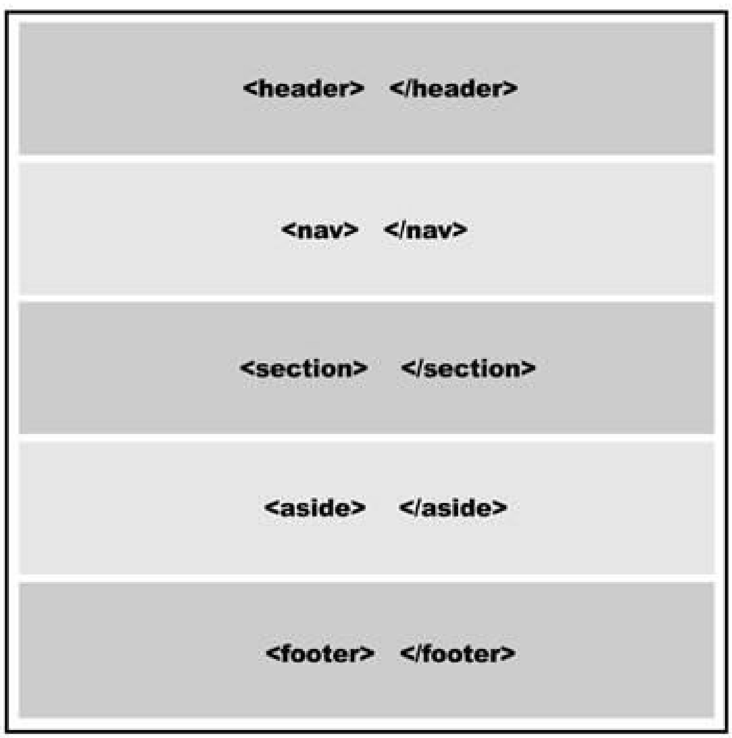


Figura 2-1. *Representación visual de una página web mostrada con estilos por defecto.*

Modelos de caja

Para aprender cómo podemos crear nuestra propia organización de los elementos en pantalla, debemos primero entender cómo los navegadores procesan el código HTML. Los navegadores consideran cada elemento HTML como una caja. Una página web es en realidad un grupo de cajas ordenadas siguiendo ciertas reglas. Estas reglas son establecidas por estilos provistos por los navegadores o por los diseñadores usando CSS.

CSS tiene un set predeterminado de propiedades destinados a sobrescribir los estilos provistos por navegadores y obtener la organización deseada. Estas propiedades no son específicas, tienen que ser combinadas para formar reglas que luego serán usadas para agrupar cajas y obtener la correcta disposición en pantalla. La combinación de estas reglas es normalmente llamada modelo o sistema de disposición. Todas estas reglas aplicadas juntas constituyen lo que se llama un modelo de caja.

Existe solo un modelo de caja que es considerado estándar estos días, y muchos otros que aún se encuentran en estado experimental. El modelo válido y ampliamente adoptado es el llamado Modelo de Caja Tradicional, el cual ha sido usado desde la primera versión de CSS.

Aunque este modelo ha probado ser efectivo, algunos modelos experimentales intentan superar sus deficiencias, pero la falta de consenso sobre el reemplazo más adecuado aún mantiene a este viejo modelo en vigencia y la mayoría de los sitios webs programados en HTML5 lo continúan utilizando.

2.3 Conceptos básicos sobre estilos

Antes de comenzar a insertar reglas CSS en nuestro archivo de estilos y aplicar un modelo de caja, debemos revisar los conceptos básicos sobre estilos CSS que van a ser utilizados en el resto del libro.

Aplicar estilos a los elementos HTML cambia la forma en que estos son presentados en pantalla. Como explicamos anteriormente, los navegadores proveen estilos por defecto que en la mayoría de los casos no son suficientes para satisfacer las necesidades de los diseñadores. Para cambiar esto, podemos sobrescribir estos estilos con los nuestros usando diferentes técnicas.

> **Conceptos básicos:** En este libro encontrará solo una introducción breve a los estilos CSS. Solo mencionamos las técnicas y propiedades que necesita conocer para entender los temas y códigos estudiados en próximos capítulos. Si considera que no tiene la suficiente experiencia en CSS y necesita mayor información visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

> **Hágalo usted mismo:** Dentro de un archivo de texto vacío, copie cada código HTML estudiado en los siguientes listados y abra el archivo en su navegador para comprobar su funcionamiento. Tenga en cuenta que el archivo debe tener la extensión **.html** para ser abierto y procesado correctamente.

Estilos en línea

Una de las técnicas más simples para incorporar estilos CSS a un documento HTML es la de asignar los estilos dentro de las etiquetas por medio del atributo **style**.

El Listado 2-1 muestra un documento HTML simple que contiene el elemento **<p>** modificado por el atributo **style** con el valor **font-size: 20px**. Este estilo cambia el tamaño por defecto del texto dentro del elemento **<p>** a un nuevo tamaño de 20 pixeles.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este es el título del documento</title>
</head>
<body>
  <p style=”font-size: 20px”>Mi texto</p>
</body>
</html>
```

Listado 2-1. *Estilos CSS dentro de etiquetas HTML.*

Usar la técnica demostrada anteriormente es una buena manera de probar estilos y obtener una vista rápida de sus efectos, pero no es recomendado para aplicar estilos a todo el documento. La razón es simple: cuando usamos esta técnica, debemos escribir y repetir cada estilo en cada uno de los elementos que queremos modificar, incrementando el tamaño del documento a proporciones inaceptables y haciéndolo imposible de mantener y actualizar. Solo imagine lo que ocurriría si decide que en lugar de 20 pixeles el tamaño de cada uno de los elementos **<p>** debería ser de 24 pixeles. Tendría que modificar cada estilo en cada etiqueta **<p>** en el documento completo.

Estilos embebidos

Una mejor alternativa es insertar los estilos en la cabecera del documento y luego usar referencias para afectar los elementos HTML correspondientes:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <style>
    p { font-size: 20px }
  </style>
```

```
</head>
<body>
  <p>Mi texto</p>
</body>
</html>
```

Listado 2-2. *Estilos listados en la cabecera del documento.*

El elemento **<style>** (mostrado en el Listado 2-2) permite a los desarrolladores agrupar estilos CSS dentro del documento. En versiones previas de HTML era necesario especificar qué tipo de estilos serían insertados. En HTML5 los estilos por defecto son CSS, por lo tanto no necesitamos agregar ningún atributo en la etiqueta de apertura **<style>**.

El código resaltado del Listado 2-2 tiene la misma función que la línea de código del Listado 2-1, pero en el Listado 2-2 no tuvimos que escribir el estilo dentro de cada etiqueta **<p>** porque todos los elementos **<p>** ya fueron afectados. Con este método, reducimos nuestro código y asignamos los estilos que queremos a elementos específicos utilizando referencias. Veremos más sobre referencias en este capítulo.

Archivos externos

Declarar los estilos en la cabecera del documento ahorra espacio y vuelve al código más consistente y actualizable, pero nos requiere hacer una copia de cada grupo de estilos en todos los documentos de nuestro sitio web. La solución es mover todos los estilos a un archivo externo y luego utilizar el elemento **<link>** para insertar este archivo dentro de cada documento que los necesite. Este método nos permite cambiar los estilos por completo simplemente incluyendo un archivo diferente. También nos permite modificar o adaptar nuestros documentos a cada circunstancia o dispositivo, como veremos al final del libro.

En el Capítulo 1, estudiamos la etiqueta **<link>** y cómo utilizarla para insertar archivos con estilos CSS en nuestros documentos. Utilizando la línea **<link rel=”stylesheet” href=”misestilos.css”>** le decimos al navegador que cargue el archivo **misestilos.css** porque contiene todos los estilos necesitados para presentar el documento en pantalla. Esta práctica fue ampliamente adoptada por diseñadores que ya están trabajando con HTML5. La etiqueta **<link>** referenciando el archivo CSS será insertada en cada uno de los documentos que requieren de esos estilos:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <p>Mi texto</p>
</body>
</html>
```

Listado 2-3. *Aplicando estilos CSS desde un archivo externo.*

Hágalo usted mismo: De ahora en adelante agregaremos estilos CSS al archivo llamado **misestilos.css**. Debe crear este archivo en el mismo directorio (carpeta) donde se encuentra el archivo HTML y copiar los estilos CSS en su interior para comprobar cómo trabajan.

Conceptos básicos: Los archivos CSS son archivos de texto comunes. Al igual que los archivos HTML, puede crearlos utilizando cualquier editor de texto como el Bloc de Notas de Windows, por ejemplo.

Referencias

Almacenar todos nuestros estilos en un archivo externo e insertar este archivo dentro de cada documento que lo necesite es muy conveniente, sin embargo no podremos hacerlo sin buenos mecanismos que nos ayuden a establecer una específica relación entre estos estilos y los elementos del documento que van a ser afectados.

Cuando hablábamos sobre cómo incluir estilos en el documento, mostramos una de las técnicas utilizadas a menudo en CSS para referenciar elementos HTML. En el Listado 2-2, el estilo para cambiar el tamaño de la letra referenciaba cada elemento **<p>** usando la palabra clave **p**. De esta manera el estilo insertado entre las etiquetas **<style>** referenciaba cada etiqueta **<p>** del documento y asignaba ese estilo particular a cada una de ellas.

Existen varios métodos para seleccionar cuáles elementos HTML serán afectados por las reglas CSS:

- referencia por la palabra clave del elemento
- referencia por el atributo **`id`**
- referencia por el atributo **`class`**

Más tarde veremos que CSS3 es bastante flexible a este respecto e incorpora nuevas y más específicas técnicas para referenciar elementos, pero por ahora aplicaremos solo estas tres.

Referenciando con palabra clave

Al declarar las reglas CSS utilizando la palabra clave del elemento afectamos cada elemento de la misma clase en el documento. Por ejemplo, la siguiente regla cambiará los estilos de todos los elementos **`<p>`**:

```
p { font-size: 20px }
```

Listado 2-4. *Referenciando por palabra clave.*

Esta es la técnica presentada previamente en el Listado 2-2. Utilizando la palabra clave **`p`** al frente de la regla le estamos diciendo al navegador que esta regla debe ser aplicada a cada elemento **`<p>`** encontrado en el documento HTML. Todos los textos envueltos en etiquetas **`<p>`** tendrán el tamaño de 20 pixeles.

Por supuesto, lo mismo funcionará para cualquier otro elemento HTML. Si especificamos la palabra clave **`span`** en lugar de **`p`**, por ejemplo, cada texto entre etiquetas **`<span>`** tendrá un tamaño de 20 pixeles:

```
span { font-size: 20px }
```

Listado 2-5. *Referenciando por otra palabra clave.*

¿Pero qué ocurre si solo necesitamos referenciar una etiqueta específica? ¿Debemos usar nuevamente el atributo **`style`** dentro de esta etiqueta? La respuesta es no. Como aprendimos anteriormente, el método de **Estilos en Línea** (usando el atributo **`style`** dentro de etiquetas HTML) es una técnica en desuso y debería ser evitada. Para seleccionar un elemento HTML específico desde las reglas de nuestro archivo CSS, podemos usar dos atributos diferentes: **`id`** y **`class`**.

Referenciando con el atributo id

El atributo **`id`** es como un nombre que identifica al elemento. Esto significa que el valor de este atributo no puede ser duplicado. Este nombre debe ser único en todo el documento.

Para referenciar un elemento en particular usando el atributo **`id`** desde nuestro archivo CSS la regla debe ser declarada con el símbolo **`#`** al frente del valor que usamos para identificar el elemento:

```
#texto1 { font-size: 20px }
```

Listado 2-6. *Referenciando a través del valor del atributo id.*

La regla en el Listado 2-6 será aplicada al elemento HTML identificado con el atributo **`id="texto1"`**. Ahora nuestro código HTML lucirá de esta manera:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
```

```
<body>
  <p id="texto1">Mi texto</p>
</body>
</html>
```

Listado 2-7. *Identificando el elemento <p> a través de su atributo id.*

El resultado de este procedimiento es que cada vez que hacemos una referencia usando el identificador **texto1** en nuestro archivo CSS, el elemento con ese valor de **id** será modificado, pero el resto de los elementos **<p>**, o cualquier otro elemento en el mismo documento, no serán afectados.

Esta es una forma extremadamente específica de referenciar un elemento y es normalmente utilizada para elementos más generales, como etiquetas estructurales. El atributo **id** y su especificidad es de hecho más apropiado para referencias en Javascript, como veremos en próximos capítulos.

Referenciando con el atributo class

La mayoría del tiempo, en lugar de utilizar el atributo **id** para propósitos de estilos es mejor utilizar **class**. Este atributo es más flexible y puede ser asignado a cada elemento HTML en el documento que comparte un diseño similar:

```
.texto1 { font-size: 20px }
```

Listado 2-8. *Referenciando por el valor del atributo class.*

Para trabajar con el atributo **class,** debemos declarar la regla CSS con un punto antes del nombre. La ventaja de este método es que insertar el atributo **class** con el valor **texto1** será suficiente para asignar estos estilos a cualquier elemento que queramos:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <p class="texto1">Mi texto</p>
  <p class="texto1">Mi texto</p>
  <p>Mi texto</p>
</body>
</html>
```

Listado 2-9. *Asignando estilos a varios elementos a través del atributo class.*

Los elementos **<p>** en las primeras dos líneas dentro del cuerpo del código en el Listado 2-9 tienen el atributo **class** con el valor **texto1**. Como dijimos previamente, la misma regla puede ser aplicada a diferentes elementos en el mismo documento. Por lo tanto, estos dos primeros elementos comparten la misma regla y ambos serán afectados por el estilo del Listado 2-8. El último elemento **<p>** conserva los estilos por defecto otorgados por el navegador.

La razón por la que debemos utilizar un punto delante del nombre de la regla es que es posible construir referencias más complejas. Por ejemplo, se puede utilizar el mismo valor para el atributo **class** en diferentes elementos pero asignar diferentes estilos para cada tipo:

```
p.texto1 { font-size: 20px }
```

Listado 2-10. *Referenciando solo elementos <p> a través del valor del atributo class.*

En el Listado 2-10 creamos una regla que referencia la clase llamada **texto1** pero solo para los elementos de tipo **<p>**. Si cualquier otro elemento tiene el mismo valor en su atributo **class** no será modificado por esta regla en particular.

Referenciando con cualquier atributo

Aunque los métodos de referencia estudiados anteriormente cubren un variado espectro de situaciones, a veces no son suficientes para encontrar el elemento exacto. La última versión de CSS ha incorporado nuevas formas de referenciar elementos HTML. Uno de ellas es el **Selector de Atributo**. Ahora podemos referenciar un elemento no solo por los atributos **id** y **class** sino también a través de cualquier otro atributo:

```
p[name] { font-size: 20px }
```

Listado 2-11. *Referenciando solo elementos <p> que tienen el atributo name.*

La regla en el Listado 2-11 cambia solo elementos **<p>** que tienen un atributo llamado **name**. Para imitar lo que hicimos previamente con los atributos **id** y **class**, podemos también especificar el valor del atributo:

```
p[name="mitexto"] { font-size: 20px }
```

Listado 2-12. *Referenciando elementos <p> que tienen un atributo name con el valor mitexto.*

CSS3 permite combinar "**=**" con otros para hacer una selección más específica:

```
p[name^="mi"] { font-size: 20px }
p[name$="mi"] { font-size: 20px }
p[name*="mi"] { font-size: 20px }
```

Listado 2-13. *Nuevos selectores en CSS3.*

Si usted conoce **Expresiones Regulares** desde otros lenguajes como Javascript o PHP, podrá reconocer los selectores utilizados en el Listado 2-13. En CSS3 estos selectores producen similares resultados:

- La regla con el selector **^=** será asignada a todo elemento **<p>** que contiene un atributo **name** con un valor comenzado en "**mi**" (por ejemplo, "**mitexto**", "**micasa**").
- La regla con el selector **$=** será asignada a todo elemento **<p>** que contiene un atributo **name** con un valor finalizado en "**mi**" (por ejemplo "**textomi**", "**casami**").
- La regla con el selector ***=** será asignada a todo elemento **<p>** que contiene un atributo **name** con un valor que incluye el texto "**mi**" (en este caso, el texto podría también encontrarse en el medio, como en "**textomicasa**").

En estos ejemplos usamos el elemento **<p>**, el atributo **name**, y una cadena de texto al azar como "**mi**", pero la misma técnica puede ser utilizada con cualquier atributo y valor que necesitemos. Solo tiene que escribir los corchetes e insertar entre ellos el nombre del atributo y el valor que necesita para referenciar el elemento HTML correcto.

Referenciando con pseudo clases

CSS3 también incorpora nuevas pseudo clases que hacen la selección aún más específica.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
  <div id="wrapper">
    <p class="mitexto1">Mi texto1</p>
    <p class="mitexto2">Mi texto2</p>
    <p class="mitexto3">Mi texto3</p>
    <p class="mitexto4">Mi texto4</p>
  </div>
```

```
</body>
</html>
```

Listado 2-14. *Plantilla para probar pseudo clases.*

Miremos por un momento el nuevo código HTML del Listado 2-14. Contiene cuatro elementos **<p>** que, considerando la estructura HTML, son hermanos entre sí e hijos del mismo elemento **<div>**.

Usando pseudo clases podemos aprovechar esta organización y referenciar un elemento específico sin importar cuánto conocemos sobre sus atributos y el valor de los mismos:

```
p:nth-child(2){
  background: #999999;
}
```

Listado 2-15. *Pseudo clase nth-child().*

La pseudo clase es agregada usando dos puntos luego de la referencia y antes del su nombre. En la regla del Listado 2-15 referenciamos solo elementos **<p>**. Esta regla puede incluir otras referencias. Por ejemplo, podríamos escribirla como **.miclase:nth-child(2)** para referenciar todo elemento que es hijo de otro elemento y tiene el valor de su atributo **class** igual a **miclase**. La pseudo clase puede ser aplicada a cualquier tipo de referencia estudiada previamente.

La pseudo clase **nth-child()** nos permite encontrar un hijo específico. Como ya explicamos, el documento HTML del Listado 2-14 tiene cuatro elementos **<p>** que son hermanos. Esto significa que todos ellos tienen el mismo padre que es el elemento **<div>**. Lo que esta pseudo clase está realmente indicando es algo como: "el hijo en la posición…" por lo que el número entre paréntesis será el número de la posición del hijo, o índice. La regla del Listado 2-15 está referenciando cada segundo elemento **<p>** encontrado en el documento.

> **Hágalo usted mismo:** Reemplace el código en su archivo HTML por el del Listado 2-14 y abra el archivo en su navegador. Incorpore las reglas estudiadas en el Listado 2-15 dentro del archivo **misestilos.css** para comprobar su funcionamiento.

Usando este método de referencia podemos, por supuesto, seleccionar cualquier hijo que necesitemos cambiando el número de índice. Por ejemplo, la siguiente regla tendrá impacto sólo sobre el último elemento **<p>** de nuestra plantilla:

```
p:nth-child(4){
  background: #999999;
}
```

Listado 2-16. *Pseudo clase nth-child().*

Como seguramente se habrá dado cuenta, es posible asignar estilos a todos los elementos creando una regla para cada uno de ellos:

```
*{
  margin: 0px;
}

p:nth-child(1){
  background: #999999;
}
p:nth-child(2){
  background: #CCCCCC;
}
p:nth-child(3){
  background: #999999;
}
p:nth-child(4){
  background: #CCCCCC;
}
```

Listado 2-17. *Creando una lista con la pseudo clase nth-child().*

La primera regla del Listado 2-17 usa el selector universal ***** para asignar el mismo estilo a cada elemento del documento. Este nuevo selector representa cada uno de los elementos en el cuerpo del documento y es útil cuando necesitamos establecer ciertas reglas básicas. En este caso, configuramos el margen de todos los elementos en 0 pixeles para evitar espacios en blanco o líneas vacías como las creadas por el elemento **<p>** por defecto.

En el resto del código del Listado 2-17 usamos la pseudo clase **nth-child()** para generar un menú o lista de opciones que son diferenciadas claramente en la pantalla por

Hágalo usted mismo: Copie el último código dentro del archivo CSS y abra el documento HTML en su navegador para comprobar el efecto.

Para agregar más opciones al menú, podemos incorporar nuevos elementos **<p>** en el código HTML y nuevas reglas con la pseudo clase **nth-child()** usando el número de índice adecuado. Sin embargo, esta aproximación genera mucho código y resulta imposible de aplicar en sitios webs con contenido dinámico. Una alternativa para obtener el mismo resultado es aprovechar las palabras clave **odd** y **even** disponibles para esta pseudo clase:

```
*{
  margin: 0px;
}
p:nth-child(odd){
  background: #999999;
}
p:nth-child(even){
  background: #CCCCCC;
}
```

Listado 2-18. *Aprovechando las palabras clave odd y even.*

Ahora solo necesitamos dos reglas para crear la lista completa. Incluso si más adelante agregamos otras opciones, los estilos serán asignados automáticamente a cada una de ellas de acuerdo a su posición. La palabra clave **odd** para la pseudo clase **nth-child()** afecta los elementos **<p>** que son hijos de otro elemento y tienen un índice impar. La palabra clave **even**, por otro lado, afecta a aquellos que tienen un índice par.

Existen otras importantes pseudo clases relacionadas con esta última, como **first-child**, **last-child** y **only-child**, algunas de ellas recientemente incorporadas. La pseudo clase **first-child** referencia solo el primer hijo, **last-child** referencia solo el último hijo, y **only-child** afecta un elemento siempre y cuando sea el único hijo disponible. Estas pseudo clases en particular no requieren palabras clave o parámetros, y son implementadas como en el siguiente ejemplo:

```
*{
  margin: 0px;
}
p:last-child{
  background: #999999;
}
```

Listado 2-19. *Usando last-child para modificar solo el último elemento <p> de la lista.*

Otra importante pseudo clase llamada **not()** es utilizada realizar una negación:

```
:not(p){
  margin: 0px;
}
```

Listado 2-20. *Aplicando estilos a cada elemento, excepto <p>.*

La regla del Listado 2-20 asignará un margen de 0 pixeles a cada elemento del documento excepto los elementos **<p>**. A diferencia del selector universal utilizado previamente, la pseudo clase **not()** nos permite declarar una excepción. Los estilos en la regla creada con esta pseudo clase serán asignados a todo elemento excepto aquellos incluidos en la referencia entre paréntesis. En lugar de la palabra clave de un elemento podemos usar cualquier otra referencia que deseemos. En el próximo listado, por ejemplo, todos los elementos serán afectados excepto aquellos con el valor **mitexto2** en el atributo

class:

```
:not(.mitexto2){
  margin: 0px;
}
```

Listado 2-21. *Excepción utilizando el atributo class.*

Cuando aplicamos la última regla al código HTML del Listado 2-14 el navegador asigna los estilos por defecto al elemento **<p>** identificado con el atributo **class** y el valor **mitexto2** y provee un margen de 0 pixeles al resto.

Nuevos selectores

Hay algunos selectores más que fueron agregados o que ahora son considerados parte de CSS3 y pueden ser útiles para nuestros diseños. Estos selectores usan los símbolos **>**, **+** y **~** para especificar la relación entre elementos.

```
div > p.mitexto2{
  color: #990000;
}
```

Listado 2-22. *Selector >.*

El selector **>** está indicando que el elemento a ser afectado por la regla es el elemento de la derecha cuando tiene al de la izquierda como su padre. La regla en el Listado 2-22 modifica los elementos **<p>** que son hijos de un elemento **<div>**. En este caso, fuimos bien específicos y referenciamos solamente el elemento **<p>** con el valor **mitexto2** en su atributo **class**.

El próximo ejemplo construye un selector utilizando el símbolo **+**. Este selector referencia al elemento de la derecha cuando es inmediatamente precedido por el de la izquierda. Ambos elementos deben compartir el mismo padre:

```
p.mitexto2 + p{
  color: #990000;
}
```

Listado 2-23. *Selector +.*

La regla del Listado 2-23 afecta al elemento **<p>** que se encuentra ubicado luego de otro elemento **<p>** identificado con el valor **mitexto2** en su atributo **class**. Si abre en su navegador el archivo HTML con el código del Listado 2-14, el texto en el tercer elemento **<p>** aparecerá en la pantalla en color rojo debido a que este elemento **<p>** en particular está posicionado inmediatamente después del elemento **<p>** identificado con el valor **mitexto2** en su atributo **class**.

El último selector que estudiaremos es el construido con el símbolo **~**. Este selector es similar al anterior pero el elemento afectado no necesita estar precediendo de inmediato al elemento de la izquierda. Además, más de un elemento puede ser afectado:

```
p.mitexto2 ~ p{
  color: #990000;
}
```

Listado 2-24. *Selector ~.*

La regla del Listado 2-24 afecta al tercer y cuarto elemento **<p>** de nuestra plantilla de ejemplo. El estilo será aplicado a todos los elementos **<p>** que son hermanos y se encuentran luego del elemento **<p>** identificado con el valor **mitexto2** en su atributo **class**. No importa si otros elementos se encuentran intercalados, los elementos **<p>** en la tercera y cuarta posición aún serán afectados. Puede verificar esto último insertando un elemento **mitexto** luego del elemento **<p>** que tiene el valor **mitexto2** en su atributo **class**. A pesar de este cambio solo los elementos **<p>** serán modificados por esta regla.

2.4 Aplicando CSS a nuestra plantilla

Como aprendimos más temprano en este mismo capítulo, todo elemento estructural es considerado una caja y la estructura completa es presentada como un grupo de cajas. Las cajas agrupadas constituyen lo que es llamado un Modelo de Caja.

Siguiendo con los conceptos básicos de CSS, vamos a estudiar lo que es llamado el Modelo de Caja Tradicional. Este modelo has sido implementado desde la primera versión de CSS y es actualmente soportado por cada navegador en el mercado, lo que lo ha convertido en un estándar para el diseño web.

Todo modelo, incluso aquellos aún en fase experimental, pueden ser aplicados a la misma estructura HTML, pero esta estructura debe ser preparada para ser afectada por estos estilos de forma adecuada. Nuestros documentos HTML deberán ser adaptados al modelo de caja seleccionado.

> **IMPORTANTE:** El Modelo de Caja Tradicional presentado posteriormente no es una incorporación de HTML5, pero es introducido en este libro por ser el único disponible en estos momentos y posiblemente el que continuará siendo utilizado en sitios webs desarrollados en HTML5 durante los próximos años. Si usted ya conoce cómo implementarlo, siéntase en libertad de obviar esta parte del capítulo.

2.5 Modelo de caja tradicional

Todo comenzó con tablas. Las tablas fueron los elementos que sin intención se volvieron la herramienta ideal utilizada por desarrolladores para crear y organizar cajas de contenido en la pantalla. Este puede ser considerado el primer modelo de caja de la web. Las cajas eran creadas expandiendo celdas y combinando filas de celdas, columnas de celdas y tablas enteras, unas sobre otras o incluso anidadas. Cuando los sitios webs crecieron y se volvieron más y más complejos esta práctica comenzó a presentar serios problemas relacionados con el tamaño y el mantenimiento del código necesario para crearlos.

Estos problemas iniciales hicieron necesario lo que ahora vemos como una práctica natural: la división entre estructura y presentación. Usando etiquetas **<div>** y estilos CSS fue posible reemplazar la función de tablas y efectivamente separar la estructura HTML de la presentación. Con elementos **<div>** y CSS podemos crear cajas en la pantalla, posicionar estas cajas a un lado o a otro y darles un tamaño, color o borde específico entre otras características. CSS provee propiedades específicas que nos permiten organizar las cajas acorde a nuestros deseos. Estas propiedades son lo suficientemente poderosas como para crear un modelo de caja que se transformó en lo que hoy conocemos como Modelo de Caja Tradicional.

Algunas deficiencias en este modelo mantuvieron a las tablas vivas por algún tiempo, pero los principales desarrolladores, influenciados por el suceso de las implementaciones Ajax y una cantidad enorme de nuevas aplicaciones interactivas, gradualmente volvieron a las etiquetas **<div>** y estilos CSS en un estándar. Finalmente el Modelo de Caja Tradicional fue adoptado a gran escala.

Plantilla

En el Capítulo 1 construimos una plantilla HTML5. Esta plantilla tiene todos los elementos necesarios para proveer estructura a nuestro documento, pero algunos detalles deben ser agregados para aplicar los estilos CSS y el Modelo de Caja Tradicional.

Este modelo necesita agrupar cajas juntas para ordenarlas horizontalmente. Debido a que el contenido completo del cuerpo es creado a partir de cajas, debemos agregar un elemento **<div>** para agruparlas, centrarlas y darles un tamaño específico.

La nueva plantilla lucirá de este modo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="iso-8859-1">
  <meta name="description" content="Ejemplo de HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Este texto es el título del documento</title>
  <link rel="stylesheet" href="misestilos.css">
</head>
<body>
<div id="agrupar">
  <header id="cabecera">
    <h1>Este es el título principal del sitio web</h1>
  </header>
  <nav id="menu">
    <ul>
      <li>principal</li>
      <li>fotos</li>
      <li>videos</li>
      <li>contacto</li>
    </ul>
  </nav>
  <section id="seccion">
    <article>
      <header>
        <hgroup>
          <h1>Título del mensaje uno</h1>
          <h2>Subtítulo del mensaje uno</h2>
        </hgroup>
```

```
        <time datetime="2011-12-10" pubdate>publicado 10-12-2011
        </time>
      </header>
      Este es el texto de mi primer mensaje
      <figure>
        <img src="http://minkbooks.com/content/myimage.jpg">
        <figcaption>
          Esta es la imagen del primer mensaje
        </figcaption>
      </figure>
      <footer>
        <p>comentarios (0)</p>
      </footer>
    </article>
    <article>
      <header>
        <hgroup>
          <h1>Título del mensaje dos</h1>
          <h2>Subtítulo del mensaje dos</h2>
        </hgroup>
        <time datetime="2011-12-15" pubdate>publicado 15-12-2011
        </time>
      </header>
      Este es el texto de mi segundo mensaje
      <footer>
        <p>comentarios (0)</p>
      </footer>
    </article>
  </section>
  <aside id="columna">
    <blockquote>Mensaje número uno</blockquote>
    <blockquote>Mensaje número dos</blockquote>
  </aside>
  <footer id="pie">
    Derechos Reservados &copy; 2010-2011
  </footer>
</div>
</body>
</html>
```

Listado 2-25. *Nueva plantilla HTML5 lista para estilos CSS.*

El Listado 2-25 provee una nueva plantilla lista para recibir los estilos CSS. Dos cambios importantes pueden distinguirse al comparar este código con el del Listado 1-18 del Capítulo 1. El primero es que ahora varias etiquetas fueron identificadas con los atributos **id** y **class**. Esto significa que podemos referenciar un elemento específico desde las reglas CSS con el valor de su atributo **id** o podemos modificar varios elementos al mismo tiempo usando el valor de su atributo **class**.

El segundo cambio realizado a la vieja plantilla es la adición del elemento **<div>** mencionado anteriormente. Este **<div>** fue identificado con el atributo y el valor **id=”agrupar”**, y es cerrado al final del cuerpo con la etiqueta de cierre **</div>**. Este elemento se encarga de agrupar todos los demás elementos permitiéndonos aplicar el modelo de caja al cuerpo y designar su posición horizontal, como veremos más adelante.

> **Hágalo usted mismo:** Compare el código del Listado 1-18 del Capítulo 1 con el código en el Listado 2-25 de este capítulo y ubique las etiquetas de apertura y cierre del elemento **<div>** utilizado para agrupar al resto. También compruebe cuáles elementos se encuentran ahora identificados con el atributo **id** y cuáles con el atributo **class**. Confirme que los valores de los atributos **id** son únicos para cada etiqueta. También necesitará reemplazar el código en el archivo HTML creado anteriormente por el del Listado 2-25 para aplicar los siguientes estilos CSS.

Con el documento HTML finalizado es tiempo de trabajar en nuestro archivo de estilos.

Selector universal *

Comencemos con algunas reglas básicas que nos ayudarán a proveer consistencia al diseño:

```
* {
  margin: 0px;
  padding: 0px;
}
```

Listado 2-26. *Regla CSS general.*

Normalmente, para la mayoría de los elementos, necesitamos personalizar los márgenes o simplemente mantenerlos al mínimo. Algunos elementos por defecto tienen márgenes que son diferentes de cero y en la mayoría de los casos demasiado amplios. A medida que avanzamos en la creación de nuestro diseño encontraremos que la mayoría de los elementos utilizados deben tener un margen de 0 pixeles. Para evitar el tener que repetir estilos constantemente, podemos utilizar el selector universal.

La primera regla en nuestro archivo CSS, presentada en el Listado 2-26, nos asegura que todo elemento tendrá un margen interno y externo de 0 pixeles. De ahora en más solo necesitaremos modificar los márgenes de los elementos que queremos que sean mayores que cero.

Conceptos básicos: Recuerde que en HTML cada elemento es considerado como una caja. El margen (**`margin`**) es en realidad el espacio alrededor del elemento, el que se encuentra por fuera del borde de esa caja (el estilo **`padding`**, por otro lado, es el espacio alrededor del contenido del elemento pero dentro de sus bordes, como el espacio entre el título y el borde de la caja virtual formada por el elemento **`<h1>`** que contiene ese título). El tamaño del margen puede ser definido por lados específicos del elemento o todos sus lados a la vez. El estilo **`margin: 0px`** en nuestro ejemplo establece un margen 0 o nulo para cada elemento de la caja. Si el tamaño hubiese sido especificado en 5 pixeles, por ejemplo, la caja tendría un espacio de 5 pixeles de ancho en todo su contorno. Esto significa que la caja estaría separada de sus vecinas por 5 pixeles. Volveremos sobre este tema más adelante en este capítulo.

Hágalo usted mismo: Debemos escribir todas las reglas necesarias para otorgar estilo a nuestra plantilla en un archivo CSS. El archivo ya fue incluido dentro del código HTML por medio de la etiqueta **`<link>`**, por lo que lo único que tenemos que hacer es crear un archivo de texto vacío con nuestro editor de textos preferido, grabarlo con el nombre **`misestilos.css`** y luego copiar en su interior la regla del Listado 2-26 y todas las presentadas a continuación.

Nueva jerarquía para cabeceras

En nuestra plantilla usamos elementos **`<h1>`** y **`<h2>`** para declarar títulos y subtítulos de diferentes secciones del documento. Los estilos por defecto de estos elementos se encuentran siempre muy lejos de lo que queremos y además en HTML5 podemos reconstruir la jerarquía H varias veces en cada sección (como aprendimos en el capítulo anterior). El elemento **`<h1>`**, por ejemplo, será usado varias veces en el documento, no solo para el título principal de la página web como pasaba anteriormente sino también para secciones internas, por lo que tenemos que otorgarle los estilos apropiados:

```
h1 {
  font: bold 20px verdana, sans-serif;
}
h2 {
  font: bold 14px verdana, sans-serif;
}
```

Listado 2-27. *Agregando estilos para los elementos <h1> y <h2>.*

La propiedad **`font`**, asignada a los elementos **`<h1>`** y **`<h2>`** en el Listado 2-27, nos permite declarar todos los estilos para el texto en una sola línea. Las propiedades que pueden ser declaradas usando **`font`** son: **`font-style`**, **`font-variant`**, **`font-weight`**, **`font-size/line-height`**, y **`font-family`** en este orden. Con estas reglas estamos cambiando el grosor, tamaño y tipo de letra del texto dentro de los elementos **`<h1>`** y **`<h2>`** a los valores que deseamos.

Declarando nuevos elementos HTML5

Otra regla básica que debemos declarar desde el comienzo es la definición por defecto de elementos estructurales de HTML5. Algunos navegadores aún no reconocen estos elementos o los tratan como elementos *inline* (en línea). Necesitamos declarar los nuevos elementos HTML5 como elementos *block* para asegurarnos de que serán tratados como regularmente se hace con elementos **`<div>`** y de este modo construir nuestro modelo de caja:

```
header, section, footer, aside, nav, article, figure, figcaption,
hgroup{
  display: block;
}
```

Listado 2-28. *Regla por defecto para elementos estructurales de HTML5.*

Apartir de ahora, los elementos afectados por la regla del Listado 2-28 serán posicionados uno sobre otro a menos que especifiquemos algo diferente más adelante.

Centrando el cuerpo

El primer elemento que es parte del modelo de caja es siempre **<body>**. Normalmente, por diferentes razones de diseño, el contenido de este elemento debe ser posicionado horizontalmente. Siempre deberemos especificar el tamaño de este contenido, o un tamaño máximo, para obtener un diseño consistente a través de diferentes configuraciones de pantalla.

```
body {
  text-align: center;
}
```

Listado 2-29. *Centrando el cuerpo.*

Por defecto, la etiqueta **<body>** (como cualquier otro elemento *block*) tiene un valor de ancho establecido en 100%. Esto significa que el cuerpo ocupará el ancho completo de la ventana del navegador. Por lo tanto, para centrar la página en la pantalla necesitamos centrar el contenido dentro del cuerpo. Con la regla agregada en el Listado 2-29, todo lo que se encuentra dentro de **<body>** será centrado en la ventana, centrando de este modo toda la página web.

Creando la caja principal

Siguiendo con el diseño de nuestra plantilla, debemos especificar una tamaño o tamaño máximo para el contenido del cuerpo. Como seguramente recuerda, en el Listado 2-25 en este mismo capítulo agregamos un elemento **<div>** a la plantilla para agrupar todas las cajas dentro del cuerpo. Este **<div>** será considerado la caja principal para la construcción de nuestro modelo de caja (este es el propósito por el que lo agregamos). De este modo, modificando el tamaño de este elemento lo hacemos al mismo tiempo para todos los demás:

```
#agrupar {
  width: 960px;
  margin: 15px auto;
  text-align: left;
}
```

Listado 2-30. *Definiendo las propiedades de la caja principal.*

La regla en el Listado 2-30 está referenciando por primera vez un elemento a través del valor de su atributo **id**. El carácter **#** le está diciendo al navegador que el elemento afectado por este conjunto de estilos tiene el atributo **id** con el valor **agrupar**.

Esta regla provee tres estilos para la caja principal. El primer estilo establece un valor fijo de 960 pixeles. Esta caja tendrá siempre un ancho de 960 pixeles, lo que representa un valor común para un sitio web estos días (los valores se encuentran entre 960 y 980 pixeles de ancho, sin embargo estos parámetros cambian constantemente a través del tiempo, por supuesto).

El segundo estilo es parte de lo que llamamos el Modelo de Caja Tradicional. En la regla previa (Listado 2-29), especificamos que el contenido del cuerpo sería centrado horizontalmente con el estilo **text-align: center**. Pero esto solo afecta contenido *inline*, como textos o imágenes. Para elementos *block*, como un **<div>**, necesitamos establecer un valor específico para sus márgenes que los adapta automáticamente al tamaño de su elemento padre. La propiedad **margin** usada para este propósito puede tener cuatro valores: superior, derecho, inferior, izquierdo, en este orden. Esto significa que el primer valor declarado en el estilo representa el margen de la parte superior del elemento, el segundo es el margen de la derecha, y así sucesivamente. Sin embargo, si solo escribimos los primeros dos parámetros, el resto tomará los mismos

valores. En nuestro ejemplo estamos usando esta técnica.

En el Listado 2-30, el estilo **margin: 15px auto** asigna 15 pixeles al margen superior e inferior del elemento **<div>** que está afectando y declara como automático el tamaño de los márgenes de izquierda y derecha (los dos valores declarados son usados para definir los cuatro márgenes). De esta manera, habremos generado un espacio de 15 pixeles en la parte superior e inferior del cuerpo y los espacios a los laterales (margen izquierdo y derecho) serán calculados automáticamente de acuerdo al tamaño del cuerpo del documento y el elemento **<div>**, efectivamente centrando el contenido en pantalla.

La página web ya está centrada y tiene un tamaño fijo de 960 pixeles. Lo próximo que necesitamos hacer es prevenir un problema que ocurre en algunos navegadores. La propiedad **text-align** es hereditaria. Esto significa que todos los elementos dentro del cuerpo y su contenido serán centrados, no solo la caja principal. El estilo asignado a **<body>** en el Listado 2-29 será asignado a cada uno de sus hijos. Debemos retornar este estilo a su valor por defecto para el resto del documento. El tercer y último estilo incorporado en la regla del Listado 2-30 (**text-align: left**) logra este propósito. El resultado final es que el contenido del cuerpo es centrado pero el contenido de la caja principal (el **<div>** identificado como **agrupar**) es alineado nuevamente hacia la izquierda, por lo tanto todo el resto del código HTML dentro de esta caja hereda este estilo.

Hágalo usted mismo: Si aún no lo ha hecho, copie cada una de las reglas listadas hasta este punto dentro de un archivo de texto vacío llamado **misestilos.css**. Este archivo debe estar ubicado en el mismo directorio (carpeta) que el archivo HTML con el código del Listado 2-25. Al terminar, deberá contar con dos archivos, uno con el código HTML y otro llamado **misestilos.css** con todos los estilos CSS estudiados desde el Listado 2-26. Abra el archivo HTML en su navegador y en la pantalla podrá notar la caja creada.

La cabecera

Continuemos con el resto de los elementos estructurales. Siguiendo la etiqueta de apertura del **<div>** principal se encuentra el primer elemento estructural de HTML5: **<header>**. Este elemento contiene el título principal de nuestra página web y estará ubicado en la parte superior de la pantalla. En nuestra plantilla, **<header>** fue identificado con el atributo **id** y el valor **cabecera**.

Como ya mencionamos, cada elemento *block*, así como el cuerpo, por defecto tiene un valor de ancho del 100%. Esto significa que el elemento ocupará todo el espacio horizontal disponible. En el caso del cuerpo, ese espacio es el ancho total de la pantalla visible (la ventana del navegador), pero en el resto de los elementos el espacio máximo disponible estará determinado por el ancho de su elemento padre. En nuestro ejemplo, el espacio máximo disponible para los elementos dentro de la caja principal será de 960 pixeles, porque su padre es la caja principal la cual fue previamente configurada con este tamaño.

```
#cabecera {
  background: #FFFBB9;
  border: 1px solid #999999;
  padding: 20px;
}
```

Listado 2-31. *Agregando estilos para <header>.*

Debido a que **<header>** ocupará todo el espacio horizontal disponible en la caja principal y será tratado como un elemento *block* (y por esto posicionada en la parte superior de la página), lo único que resta por hacer es asignar estilos que nos permitirán reconocer el elemento cuando es presentado en pantalla. En la regla mostrada en el Listado 2-31 le otorgamos a **<header>** un fondo amarillo, un borde sólido de 1 pixel y un margen interior de 20 pixeles usando la propiedad **padding**.

Barra de navegación

Siguiendo al elemento **<header>** se encuentra el elemento **<nav>**, el cual tiene el propósito de proporcionar ayuda para la navegación. Los enlaces agrupados dentro de este elemento representarán el menú de nuestro sitio web. Este menú será una simple barra ubicada debajo de la cabecera. Por este motivo, del mismo modo que el elemento **<header>**, la mayoría de los estilos que necesitamos para posicionar el elemento **<nav>** ya fueron asignados: **<nav>** es un elemento *block* por lo que será ubicado debajo del elemento previo, su ancho por defecto será 100% por lo que será tan ancho como su padre (el **<div>** principal), y (también por defecto) será tan alto como su contenido y los márgenes predeterminados. Por lo tanto, lo único que nos queda por hacer es mejorar su aspecto en pantalla. Esto último lo logramos agregando un fondo gris y un pequeño margen interno para separar las opciones del menú del borde del elemento:

```
#menu {
  background: #CCCCCC;
  padding: 5px 15px;
}
#menu li {
  display: inline-block;
  list-style: none;
  padding: 5px;
  font: bold 14px verdana, sans-serif;
}
```

Listado 2-32. *Agregando estilos para <nav>.*

En el Listado 2-32, la primera regla referencia al elemento **<nav>** por su atributo **id**, cambia su color de fondo y agrega márgenes internos de **5px** y **15px** con la propiedad **padding**.

> **Conceptos básicos:** La propiedad **padding** trabaja exactamente como **margin**. Cuatro valores pueden ser especificados: superior, derecho, inferior, izquierdo, en este orden. Si solo declaramos un valor, el mismo será asignado para todos los espacios alrededor del contenido del elemento. Si en cambio especificamos dos valores, entonces el primero será asignado como margen interno de la parte superior e inferior del contenido y el segundo valor será asignado al margen interno de los lados, izquierdo y derecho.

Dentro de la barra de navegación hay una lista creada con las etiquetas **<ul>** y **<li>**. Por defecto, los ítems de una lista son posicionados unos sobre otros. Para cambiar este comportamiento y colocar cada opción del menú una al lado de la otra, referenciamos los elementos **<li>** dentro de este elemento **<nav>** en particular usando el selector **#menu li**, y luego asignamos a todos ellos el estilo **display: inline-block** para convertirlos en lo que se llama cajas *inline*. A diferencia de los elementos *block*, los elementos afectados por el parámetro **inline-block** estandarizado en CSS3 no generan ningún salto de línea pero nos permiten tratarlos como elementos block y así declarar un valor de ancho determinado. Este parámetro también ajusta el tamaño del elemento de acuerdo con su contenido cuando el valor del ancho no fue especificado.

En esta última regla también eliminamos el pequeño gráfico generado por defecto por los navegadores delante de cada opción del listado utilizando la propiedad **list-style**.

Section y aside

Los siguientes elementos estructurales en nuestro código son dos cajas ordenadas horizontalmente. El Modelo de Caja Tradicional es construido sobre estilos CSS que nos permiten especificar la posición de cada caja. Usando la propiedad **float** podemos posicionar estas cajas del lado izquierdo o derecho de acuerdo a nuestras necesidades. Los elementos que utilizamos en nuestra plantilla HTML para crear estas cajas son **<section>** y **<aside>**, cada uno identificado con el atributo **id** y los valores **seccion** y **columna** respectivamente.

```
#seccion {
  float: left;
  width: 660px;
  margin: 20px;
}
#columna {
  float: left;
  width: 220px;
  margin: 20px 0px;
  padding: 20px;
  background: #CCCCCC;
}
```

Listado 2-33. *Creando dos columnas con la propiedad float.*

La propiedad de CSS **float** es una de las propiedades más ampliamente utilizadas para aplicar el Modelo de Caja Tradicional. Hace que el elemento flote hacia un lado o al otro en el espacio disponible. Los elementos afectados por **float** actúan como elementos *block* (con la diferencia de que son ubicados de acuerdo al valor de esta propiedad y no el flujo normal del documento). Los elementos son movidos a izquierda o derecha en el área disponible, tanto como sea posible,

respondiendo al valor de **float**.

Con las reglas del Listado 2-33 declaramos la posición de ambas cajas y sus respectivos tamaños, generando así las columnas visibles en la pantalla. La propiedad **float** mueve la caja al espacio disponible del lado especificado por su valor, **width** asigna un tamaño horizontal y **margin**, por supuesto, declara el margen del elemento.

Afectado por estos valores, el contenido del elemento **<section>** estará situado a la izquierda de la pantalla con un tamaño de 660 pixeles, más 40 pixeles de margen, ocupando un espacio total de 700 pixeles de ancho.

La propiedad **float** del elemento **<aside>** también tiene el valor **left** (izquierda). Esto significa que la caja generada será movida al espacio disponible a su izquierda. Debido a que la caja previa creada por el elemento **<section>** fue también movida a la izquierda de la pantalla, ahora el espacio disponible será solo el que esta caja dejó libre. La nueva caja quedará ubicada en la misma línea que la primera pero a su derecha, ocupando el espacio restante en la línea, creando la segunda columna de nuestro diseño.

El tamaño declarado para esta segunda caja fue de 220 pixeles. También agregamos un fondo gris y configuramos un margen interno de 20 pixeles. Como resultado final, el ancho de esta caja será de 220 pixeles más 40 pixeles agregados por la propiedad **padding** (los márgenes de los lados fueron declarados a **0px**).

> **Conceptos básicos:** El tamaño de un elemento y sus márgenes son agregados para obtener el valor real ocupado en pantalla. Si tenemos un elemento de 200 pixeles de ancho y un margen de 10 pixeles a cada lado, el área real ocupada por el elemento será de 220 pixeles. El total de 20 pixeles del margen es agregado a los 200 pixeles del elemento y el valor final es representado en la pantalla. Lo mismo pasa con las propiedades **padding** y **border**. Cada vez que agregamos un borde a un elemento o creamos un espacio entre el contenido y el borde usando **padding** esos valores serán agregados al ancho del elemento para obtener el valor real cuando el elemento es mostrado en pantalla. Este valor real es calculado con la fórmula: **tamaño + márgenes + márgenes internos + bordes**.

> **Hágalo usted mismo:** Lea el código del Listado 2-25. Controle cada regla CSS creada hasta el momento y busque en la plantilla los elementos HTML correspondientes a cada una de ellas. Siga las referencias, por ejemplo las claves de los elementos (como **h1)** y los atributos **id** (como **cabecera**), para entender cómo trabajan las referencias y cómo los estilos son asignados a cada elemento.

Footer

Para finalizar la aplicación del Modelo de Caja Tradicional, otra propiedad CSS tiene que ser aplicada al elemento **<footer>**. Esta propiedad devuelve al documento su flujo normal y nos permite posicionar **<footer>** debajo del último elemento en lugar de a su lado:

```
#pie {
  clear: both;
  text-align: center;
  padding: 20px;
  border-top: 2px solid #999999;
}
```

Listado 2-34. *Otorgando estilos a <footer> y recuperando el normal flujo del documento.*

La regla del Listado 2-34 declara un borde de 2 pixeles en la parte superior de **<footer>**, un margen interno (**padding**) de 20 pixeles, y centra el texto dentro del elemento. Así mismo, restaura el normal flujo del documento con la propiedad **clear**. Esta propiedad simplemente restaura las condiciones normales del área ocupada por el elemento, no permitiéndole posicionarse adyacente a una caja flotante. El valor usualmente utilizado es **both**, el cual significa que ambos lados del elemento serán restaurados y el elemento seguirá el flujo normal (este elemento ya no es flotante como los anteriores). Esto, para un elemento *block*, quiere decir que será posicionado debajo del último elemento, en una nueva línea.

La propiedad **clear** también empuja los elementos verticalmente, haciendo que las cajas flotantes ocupen un área real en la pantalla. Sin esta propiedad, el navegador presenta el documento en pantalla como si los elementos flotantes no existieran y las cajas se superponen.

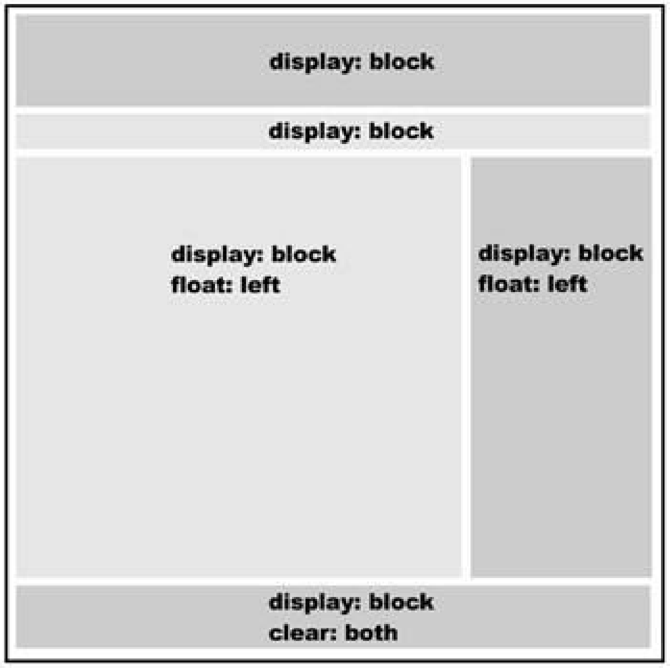


Figura 2-2. Representación visual del modelo de caja tradicional.

Cuando tenemos cajas posicionadas una al lado de la otra en el Modelo de Caja Tradicional siempre necesitamos crear un elemento con el estilo **`clear: both`** para poder seguir agregando otras cajas debajo de un modo natural. La Figura 2-2 muestra una representación visual de este modelo con los estilos básicos para lograr la correcta disposición en pantalla.

Los valores **`left`** (izquierda) y **`right`** (derecha) de la propiedad **`float`** no significan que las cajas deben estar necesariamente posicionadas del lado izquierdo o derecho de la ventana. Lo que los valores hacen es volver flotante ese lado del elemento, rompiendo el flujo normal del documento. Si el valor es **`left`**, por ejemplo, el navegador tratará de posicionar el elemento del lado izquierdo en el espacio disponible. Si hay espacio disponible luego de otro elemento, este nuevo elemento será situado a su derecha, porque su lado izquierdo fue configurado como flotante. El elemento flota hacia la izquierda hasta que encuentra algo que lo bloquea, como otro elemento o el borde de su elemento padre. Esto es importante cuando queremos crear varias columnas en la pantalla. En este caso cada columna tendrá el valor **`left`** en la propiedad **`float`** para asegurar que cada columna estará continua a la otra en el orden correcto. De este modo, cada columna flotará hacia la izquierda hasta que es bloqueada por otra columna o el borde del elemento padre.

Últimos toques

Lo único que nos queda por hacer es trabajar en el diseño del contenido. Para esto, solo necesitamos configurar los pocos elementos HTML5 restantes:

```
article {
  background: #FFFBCC;
  border: 1px solid #999999;
  padding: 20px;
  margin-bottom: 15px;
}
article footer {
  text-align: right;
}
time {
  color: #999999;
}
figcaption {
  font: italic 14px verdana, sans-serif;
}
```

Listado 2-35. *Agregando los últimos toques a nuestro diseño básico.*

La primera regla del Listado 2-35 referencia todos los elementos **<article>** y les otorga algunos estilos básicos (color de fondo, un borde sólido de 1 pixel, margen interno y margen inferior). El margen inferior de 15 pixeles tiene el propósito de separar un elemento **<article>** del siguiente verticalmente.

Cada elemento **<article>** cuenta también con un elemento **<footer>** que muestra el número de comentarios recibidos. Para referenciar un elemento **<footer>** dentro de un elemento **<article>**, usamos el selector **article footer** que significa "cada **<footer>** dentro de un **<article>** será afectado por los siguientes estilos". Esta técnica de referencia fue aplicada aquí para alinear a la derecha el texto dentro de los elementos **<footer>** de cada **<article>**.

Al final del código en el Listado 2-35 cambiamos el color de cada elemento **<time>** y diferenciamos la descripción de la imagen (insertada con el elemento **<figcaption>**) del resto del texto usando una tipo de letra diferente.

Hágalo usted mismo: Si aún no lo ha hecho, copie cada regla CSS listada en este capítulo desde el Listado 2-26, una debajo de otra, dentro del archivo **misestilos.css**, y luego abra el archivo HTML con la plantilla creada en el Listado 2-25 en su navegador. Esto le mostrará cómo funciona el Modelo de Caja Tradicional y cómo los elementos estructurales son organizados en pantalla.

IMPORTANTE: Puede acceder a estos códigos con un solo clic desde nuestro sitio web. Visite *www.minkbooks.com*.

Box-sizing

Existe una propiedad adicional incorporada en CSS3 relacionada con la estructura y el Modelo de Caja Tradicional. La propiedad **box-sizing** nos permite cambiar cómo el espacio total ocupado por un elemento en pantalla será calculado forzando a los navegadores a incluir en el ancho original los valores de las propiedades **padding** y **border**.

Como explicamos anteriormente, cada vez que el área total ocupada por un elemento es calculada, el navegador obtiene el valor final por medio de la siguiente fórmula: **tamaño + márgenes + márgenes internos + bordes**.

Por este motivo, si declaramos la propiedad **width** igual a 100 pixeles, **margin** en 20 pixeles, **padding** en 10 pixeles y **border** en 1 pixel, el área horizontal total ocupada por el elemento será: **100+40+20+2= 162** pixeles (note que tuvimos que duplicar los valores de **margin**, **padding** y **border** en la fórmula porque consideramos que los mismos fueron asignados tanto para el lado derecho como el izquierdo).

Esto significa que cada vez que declare el ancho de un elemento con la propiedad **width**, deberá recordar que el área real para ubicar el elemento en pantalla será seguramente más grande.

Dependiendo de sus costumbres, a veces podría resultar útil forzar al navegador a incluir los valores de **padding** y **border** en el tamaño del elemento. En este caso la nueva fórmula sería simplemente: **tamaño + márgenes**.

```
div {
  width: 100px;
  margin: 20px;
  padding: 10px;
  border: 1px solid #000000;

  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

Listado 2-36. *Incluyendo padding y border en el tamaño del elemento.*

La propiedad **box-sizing** puede tomar dos valores. Por defecto es configurada como **content-box**, lo que significa que los navegadores agregarán los valores de **padding** y **border** al tamaño especificado por **width** y **height**. Usando el valor **border-box** en su lugar, este comportamiento es cambiado de modo que **padding** y **border** son incluidos dentro del elemento.

El Listado 2-36 muestra la aplicación de esta propiedad en un elemento **<div>**. Este es solo un ejemplo y no vamos a usarlo en nuestra plantilla, pero puede ser útil para algunos diseñadores dependiendo de qué tan familiarizados se encuentran con los métodos tradicionales propuestos por versiones previas de CSS.

IMPORTANTE: En este momento, la propiedad **box-sizing**, al igual que otras importantes propiedades CSS3 estudiadas en próximos capítulos, se encuentra en estado experimental en algunos navegadores. Para aplicarla efectivamente a sus documentos, debe declararla con los correspondientes prefijos, como hicimos en el Listado 2-36. Los prefijos para los navegadores más comunes son los siguientes:

- **-moz-** para Firefox.
- **-webkit-** para Safari y Chrome.
- **-o-** para Opera.
- **-khtml-** para Konqueror.
- **-ms-** para Internet Explorer.
- **-chrome-** específico para Google Chrome.

2.6 Referencia rápida

En HTML5 la responsabilidad por la presentación de la estructura en pantalla está más que nunca en manos de CSS. Incorporaciones y mejoras se han hecho en la última versión de CSS para proveer mejores formas de organizar documentos y trabajar con sus elementos.

Selector de atributo y pseudo clases

CSS3 incorpora nuevos mecanismos para referenciar elementos HTML.

Selector de Atributo Ahora podemos utilizar otros atributos además de **`id`** y **`class`** para encontrar elementos en el documento y asignar estilos. Con la construcción **`palabraclave[atributo=valor]`**, podemos referenciar un elemento que tiene un atributo particular con un valor específico. Por ejemplo, **`p[name="texto"]`** referenciará cada elemento **`<p>`** con un atributo llamado **`name`** y el valor "**`texto`**". CSS3 también provee técnicas para hacer esta referencia aún más específica. Usando las siguientes combinaciones de símbolos **`^=`**, **`$=`** y **`*=`** podemos encontrar elementos que comienzan con el valor provisto, elementos que terminan con ese valor y elementos que tienen el texto provisto en alguna parte del valor del atributo. Por ejemplo, **`p[name^="texto"]`** será usado para encontrar elementos **`<p>`** que tienen un atributo llamado **`name`** con un valor que comienza por "**`texto`**".

Pseudo Clase :nth-child() Esta pseudo clase encuentra un hijo específico siguiendo la estructura de árbol de HTML. Por ejemplo, con el estilo **`span:nth-child(2)`** estamos referenciando el elemento **`<span>`** que tiene otros elementos **`<span>`** como hermanos y está localizado en la posición **`2`**. Este número es considerado el índice. En lugar de un número podemos usar las palabras clave **`odd`** y **`even`** para referenciar elementos con un índice impar o par respectivamente (por ejemplo, **`span:nth-child(odd)`**).

Pseudo Clase :first-child Esta pseudo clase es usada para referenciar el primer hijo, similar a **`:nth-child(1)`**.

Pseudo Clase :last-child Esta pseudo clase es usada para referenciar el último hijo.

Pseudo Clase :only-child Esta pseudo clase es usada para referenciar un elemento que es el único hijo disponible de un mismo elemento padre.

Pseudo Clase :not() Esta pseudo clase es usada para referenciar todo elemento excepto el declarado entre paréntesis.

Selectores

CSS3 también incorpora nuevos selectores que ayudan a llegar a elementos difíciles de referenciar utilizando otras técnicas.

Selector > Este selector referencia al elemento de la derecha cuando tiene el elemento de la izquierda como padre. Por ejemplo, **`div > p`** referenciará cada elemento **`<p>`** que es hijo de un elemento **`<div>`**.

Selector + Este selector referencia elementos que son hermanos. La referencia apuntará al elemento de la derecha cuando es inmediatamente precedido por el de la izquierda. Por ejemplo, **`span + p`** afectará a los elementos **`<p>`** que son hermanos y están ubicados luego de un elemento **`<span>`**.

Selector ~ Este selector es similar al anterior, pero en este caso el elemento de la derecha no tiene que estar ubicado inmediatamente después del de la izquierda.

Capítulo 3
Propiedades CSS3

3.1 Las nuevas reglas

La web cambió para siempre cuando unos años atrás nuevas aplicaciones desarrolladas sobre implementaciones Ajax mejoraron el diseño y la experiencia de los usuarios. La versión 2.0, asignada a la web para describir un nuevo nivel de desarrollo, representó un cambio no solo en la forma en que la información era transmitida sino también en cómo los sitios web y nuevas aplicaciones eran diseñados y construidos.

Los códigos implementados en esta nueva generación de sitios web pronto se volvieron estándar. La innovación se volvió tan importante para el éxito de cualquier proyecto en Internet que programadores desarrollaron librerías completas para superar las limitaciones y satisfacer los nuevos requerimientos de los diseñadores.

La falta de soporte por parte de los navegadores era evidente, pero la organización responsable de los estándares web no tomó las tendencias muy seriamente e intentó seguir su propio camino. Afortunadamente, algunas mentes brillantes siguieron desarrollando nuevos estándares en paralelo y pronto HTML5 nació. Luego del retorno de la calma (y algunos acuerdos de por medio), la integración entre HTML, CSS y Javascript bajo la tutela de HTML5 fue como el caballero bravo y victorioso que dirige las tropas hacia el palacio enemigo.

A pesar de la reciente agitación, esta batalla comenzó mucho tiempo atrás, con la primera especificación de la tercera versión de CSS. Cuando finalmente, alrededor del año 2005, esta tecnología fue oficialmente considerada estándar, CSS estaba listo para proveer las funciones requeridas por desarrolladores (aquellas que programadores habían creado desde años atrás usando códigos Javascript complicados de implementar y no siempre compatibles).

En este capítulo vamos a estudiar las contribuciones hechas por CSS3 a HTML5 y todas las propiedades que simplifican la vida de diseñadores y programadores.

CSS3 se vuelve loco

CSS fue siempre sobre estilo, pero ya no más. En un intento por reducir el uso de código Javascript y para estandarizar funciones populares, CSS3 no solo cubre diseño y estilos web sino también forma y movimiento. La especificación de CSS3 es presentada en módulos que permiten a la tecnología proveer una especificación estándar por cada aspecto involucrado en la presentación visual del documento. Desde esquinas redondeadas y sombras hasta transformaciones y reposicionamiento de los elementos ya presentados en pantalla, cada posible efecto aplicado previamente utilizando Javascript fue cubierto. Este nivel de cambio convierte CSS3 en una tecnología prácticamente inédita comparada con versiones anteriores.

Cuando la especificación de HTML5 fue escrita considerando CSS a cargo del diseño, la mitad de la batalla contra el resto de las especificaciones propuesta había sido ganada.

Plantilla

Las nuevas propiedades CSS3 son extremadamente poderosas y deben ser estudiadas una por una, pero para facilitar su aprendizaje vamos a aplicar todas ellas sobre la misma plantilla. Por este motivo comenzaremos por crear un documento HTML sencillo con algunos estilos básicos:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Nuevos Estilos CSS3</title>
  <link rel="stylesheet" href="nuevocss3.css">
</head>
<body>
<header id="principal">
  <span id="titulo">Estilos CSS Web 2.0</span>
</header>
</body>
</html>
```

Listado 3-1. *Una plantilla simple para probar nuevas propiedades.*

Nuestro documento solo tiene una sección con un texto breve en su interior. El elemento **`<header>`** usado en la plantilla podría ser reemplazado por **`<div>`**, **`<nav>`**, **`<section>`** o cualquier otro elemento estructural de acuerdo a la ubicación en el diseño y a su función. Luego de aplicar los estilos, la caja generada con el código del ejemplo del Listado 3-1 lucirá como una cabecera, por consiguiente decidimos usar **`<header>`** en este caso.

Debido a que el elemento **`<font>`** se encuentra en desuso en HTML5, los elementos usados para mostrar texto son normalmente **`<span>`** para líneas cortas y **`<p>`** para párrafos, entre otros. Por esta razón el texto en nuestra plantilla fue insertado usando etiquetas **`<span>`**.

Hágalo usted mismo: Use el código provisto en el Listado 3-1 como la plantilla para este capítulo. Necesitará además crear un nuevo archivo CSS llamado **`nuevocss3.css`** para almacenar los estilos estudiados de aquí en adelante.

Los siguientes son los estilos básicos requeridos por nuestro documento HTML:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
}
#titulo {
  font: bold 36px verdana, sans-serif;
}
```

Listado 3-2. *Reglas básicas CSS con las que comenzar.*

No hay nada nuevo en las reglas del Listado 3-2, solo los estilos necesarios para dar forma a la plantilla y crear una caja ancha, posicionada en el centro de la ventana, con un fondo gris, un borde y un texto grande en su interior que dice “Estilos CSS Web 2.0”.

Una de las cosas que notará sobre esta caja cuando sea mostrada en pantalla es que sus esquinas son rectas. Esto no es algo que nos agrade, ¿verdad? Puede ser un factor psicológico o no, lo cierto es que a casi nadie en este negocio le agradan las esquinas rectas. Por lo tanto, lo primero que haremos será cambiar este aspecto.

Border-radius

Por muchos años diseñadores han sufrido intentando lograr el efecto de esquinas redondeadas en las cajas de sus páginas web. El proceso era casi siempre frustrante y extenuante. Todos lo padecieron alguna vez. Si mira cualquier presentación en video de las nuevas características incorporadas en HTML5, cada vez que alguien habla sobre las propiedades de CSS3 que hacen posible generar fácilmente esquinas redondeadas, la audiencia enloquece. Esquinas redondeadas eran esa clase de cosas que nos hacían pensar: “debería ser fácil hacerlo”. Sin embargo nunca lo fue.

Esta es la razón por la que, entre todas las nuevas posibilidades e increíbles propiedades incorporadas en CSS3, la que exploraremos en primera instancia es **`border-radius`**:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
```

```
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;
}
#titulo {
  font: bold 36px verdana, sans-serif;
}
```

Listado 3-3. *Generando esquinas redondeadas.*

La propiedad **`border-radius`** en este momento es experimental por lo que debemos usar los prefijos **`-moz-`** y **`-webkit-`** para que funcionen en navegadores basados en motores Gecko y WebKit, como Firefox, Safari y Google Chrome (los prefijos fueron estudiados y aplicados en el Capítulo 2). Si todas las esquinas tienen la misma curvatura podemos utilizar un solo valor. Sin embargo, como ocurre con las propiedades **`margin`** y **`padding`**, podemos también declarar un valor diferente por cada una:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px 10px 30px 50px;
  -webkit-border-radius: 20px 10px 30px 50px;
  border-radius: 20px 10px 30px 50px;
}
#titulo {
  font: bold 36px verdana, sans-serif;
}
```

Listado 3-4. *Diferentes valores para cada esquina.*

Como puede ver en el Listado 3-4, los cuatro valores asignados a la propiedad **`border-radius`** representan diferentes ubicaciones. Recorriendo la caja en dirección de las agujas del reloj, los valores se aplicarán en el siguiente orden: esquina superior izquierda, esquina superior derecha, esquina inferior derecha y esquina inferior izquierda. Los valores son siempre dados en dirección de las agujas del reloj, comenzando por la esquina superior izquierda.

Al igual que con **`margin`** o **`padding`**, **`border-radius`** puede también trabajar solo con dos valores. El primer valor será asignado a la primera y tercera equina (superior izquierda, inferior derecha), y el segundo valor a la segunda y cuarta esquina (superior derecha, inferior izquierda).

También podemos dar forma a las esquinas declarando un segundo grupo de valores separados por una barra. Los valores a la izquierda de la barra representarán el radio horizontal mientras que los valores a la derecha representan el radio vertical. La combinación de estos valores genera una elipsis:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
```

```
  -moz-border-radius: 20px / 10px;
  -webkit-border-radius: 20px / 10px;
  border-radius: 20px / 10px;
}
#titulo {
  font: bold 36px verdana, sans-serif;
}
```

Listado 3-5. *Esquinas elípticas.*

Hágalo usted mismo: Copie dentro del archivo CSS llamado **nuevocss3.css** los estilos que quiera probar y abra el archivo HTML generado con el Listado 3-1 en su navegador para comprobar los resultados.

Box-shadow

Ahora que finalmente contamos con la posibilidad de generar bonitas esquinas para nuestras cajas podemos arriesgarnos con algo más. Otro muy buen efecto, que había sido extremadamente complicado de lograr hasta este momento, es sombras. Por años diseñadores han combinado imágenes, elementos y algunas propiedades CSS para generar sombras. Gracias a CSS3 y a la nueva propiedad **box-shadow** podremos aplicar sombras a nuestras cajas con solo una simple línea de código:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px;
  box-shadow: rgb(150,150,150) 5px 5px;
}
#titulo {
  font: bold 36px verdana, sans-serif;
}
```

Listado 3-6. *Aplicando sombra a nuestra caja.*

La propiedad **box-shadow** necesita al menos tres valores. El primero, que puede ver en la regla del Listado 3-6, es el color. Este valor fue construido aquí utilizando la función **rgb()** y números decimales, pero podemos escribirlo en números hexadecimales también, como hicimos previamente para otros parámetros en este libro.

Los siguientes dos valores, expresados en pixeles, establecen el desplazamiento de la sombra. Este desplazamiento puede ser positivo o negativo. Los valores indican, respectivamente, la distancia horizontal y vertical desde la sombra al elemento. Valores negativos posicionarán la sombra a la izquierda y arriba del elemento, mientras que valores positivos crearán la sombra a la derecha y debajo del elemento. Valores de 0 o nulos posicionarán la sombra exactamente detrás del elemento, permitiendo la posibilidad de crear un efecto difuminado a todo su alrededor.

Hágalo usted mismo: Para probar los diferentes parámetros y posibilidades con los que contamos para asignar una sombra a una caja, copie el código del Listado 3-6 dentro del archivo CSS y abra el archivo HTML con la plantilla del Listado 3-1 en su navegador. Puede experimentar cambiando los valores de la propiedad **box-shadow** y puede usar el mismo código para experimentar también con los nuevos parámetros estudiados a continuación.

La sombra que obtuvimos hasta el momento es sólida, sin gradientes o transparencias (no realmente como una sombra

suele aparecer). Existen algunos parámetros más y cambios que podemos implementar para mejorar la apariencia de la sombra.

Un cuarto valor que se puede agregar a la propiedad ya estudiada es la distancia de difuminación. Con este efecto ahora la sombra lucirá real. Puede intentar utilizar este nuevo parámetro declarando un valor de 10 pixeles a la regla del Listado 3-6, como en el siguiente ejemplo:

```
box-shadow: rgb(150,150,150) 5px 5px 10px;
```

Listado 3-7. *Agregando el valor de difuminación a `box-shadow`.*

Agregando otro valor más en pixeles al final de la propiedad desparramará la sombra. Este efecto cambia un poco la naturaleza de la sombra expandiendo el área que cubre. A pesar de que no recomendamos utilizar este efecto, puede ser aplicable en algunos diseños.

> **Hágalo usted mismo:** Intente agregar un valor de 20 pixeles al final del estilo del Listado 3-7 y combine este código con el código del Listado 3-6 para probarlo.

> **IMPORTANTE:** Siempre recuerde que en este momento las propiedades estudiadas son experimentales. Para usarlas, debe declarar cada una agregando los prefijos correspondientes, como **`-moz-`** o **`-webkit-`**, de acuerdo al navegador que usa (en este ejemplo, Firefox o Google Chrome).

El último valor posible para **`box-shadow`** no es un número sino más bien una palabra clave: **`inset`**. Esta palabra clave convierte a la sombra externa en una sombra interna, lo cual provee un efecto de profundidad al elemento afectado.

```
box-shadow: rgb(150,150,150) 5px 5px 10px inset;
```

Listado 3-8. *Sombra interna.*

El estilo en el Listado 3-8 mostrará una sombra interna alejada del borde de la caja por unos 5 pixeles y con un efecto de difuminación de 10 pixeles.

> **Hágalo usted mismo:** Los estilos de los Listados 3-7 y 3-8 son solo ejemplos. Para comprobar los efectos en su navegador debe aplicar estos cambios al grupo completo de reglas presentado en el Listado 3-6.

> **IMPORTANTE:** Las sombras no expanden el elemento o incrementan su tamaño, por lo que tendrá que controlar cuidadosamente que el espacio disponible es suficiente para que la sombra sea expuesta y correctamente dibujada en la pantalla.

Text-shadow

Ahora que conoce todo acerca de sombras probablemente estará pensando en generar una para cada elemento de su documento. La propiedad **`box-shadow`** fue diseñada especialmente para ser aplicada en cajas. Si intenta aplicar este efecto a un elemento **`<span>`**, por ejemplo, la caja invisible ocupada por este elemento en la pantalla tendrá una sombra, pero no el contenido del elemento. Para crear sombras para figuras irregulares como textos, existe una propiedad especial llamada **`text-shadow`**:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;
```

```
  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;
}
#titulo {
  font: bold 36px verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
```

Listado 3-9. *Generando una sombra para el título.*

Los valores para **`text-shadow`** son similares a los usados para **`box-shadow`**. Podemos declarar el color de la sombra, la distancia horizontal y vertical de la sombra con respecto al objeto y el radio de difuminación.

En el Listado 3-9 una sombra azul fue aplicada al título de nuestra plantilla con una distancia de 3 pixeles y un radio de difuminación de 5.

@font-face

Obtener un texto con sombra es realmente un muy buen truco de diseño, imposible de lograr con métodos previos, pero más que cambiar el texto en sí mismo solo provee un efecto tridimensional. Una sombra, en este caso, es como pintar un viejo coche, al final será el mismo coche. En este caso, será el mismo tipo de letra.

El problema con las fuentes o tipos de letra es tan viejo como la web. Usuarios regulares de la web a menudo tienen un número limitado de fuentes instaladas en sus ordenadores, usualmente estas fuentes son diferentes de un usuario a otro, y la mayoría de las veces muchos usuarios tendrán fuentes que otros no. Por años, los sitios webs solo pudieron utilizar un limitado grupo de fuentes confiables (un grupo básico que prácticamente todos los usuarios tienen instalados) y así presentar la información en pantalla.

La propiedad **`@font-face`** permite a los diseñadores proveer un archivo conteniendo una fuente específica para mostrar sus textos en la página. Ahora podemos incluir cualquier fuente que necesitemos con solo proveer el archivo adecuado:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;
}
#titulo {
  font: bold 36px MiNuevaFuente, verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
@font-face {
  font-family: 'MiNuevaFuente';
  src: url('font.ttf');
}
```

Listado 3-10. *Nueva fuente para el título.*

Hágalo usted mismo: Descargue el archivo **`font.ttf`** desde nuestro sitio web o use uno que ya posea y cópielo en el

mismo directorio (carpeta) de su archivo CSS. Para descargar el archivo, visite el siguiente enlace: *www.minkbooks.com/ content/font.ttf*. Puede obtener más fuentes similares de forma gratuita en *www.moorstation.org/typoasis/designers/steffmann/.*

IMPORTANTE: El archivo conteniendo la fuente debe encontrarse en el mismo dominio que la página web (o en el mismo ordenador, en este caso). Esta es una restricción de algunos navegadores como Firefox, por ejemplo.

La propiedad **`@font-face`** necesita al menos dos estilos para declarar la fuente y cargar el archivo. El estilo construido con la propiedad **`font-family`** especifica el nombre que queremos otorgar a esta fuente en particular, y la propiedad **`src`** indica la URL del archivo con el código correspondiente a esa fuente. En el Listado 3-10, el nombre **`MiNuevaFuente`** fue asignado a nuestro nuevo tipo de letra y el archivo **`font.ttf`** fue indicado como el archivo correspondiente a esta fuente.

Una vez que la fuente es cargada, podemos comenzar a usarla en cualquier elemento del documento simplemente escribiendo su nombre (**`MiNuevaFuente`**). En el estilo **`font`** en la regla del Listado 3-10, especificamos que el título será mostrado con la nueva fuente o las alternativas **`verdana`** y **`sans-serif`** en caso de que la fuente incorporada no sea cargada apropiadamente.

Gradiente lineal

Los gradientes son uno de los efectos más atractivos entre aquellos incorporados en CSS3. Este efecto era prácticamente imposible de implementar usando técnicas anteriores pero ahora es realmente fácil de hacer usando CSS. Una propiedad **`background`** con algunos pocos parámetros es suficiente para convertir su documento en una página web con aspecto profesional:

```
body {
  text-align: center;
}
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;

  background: -webkit-linear-gradient(top, #FFFFFF, #006699);
  background: -moz-linear-gradient(top, #FFFFFF, #006699);
}
#titulo {
  font: bold 36px MiNuevaFuente, verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
@font-face {
  font-family: 'MiNuevaFuente';
  src: url('font.ttf');
}
```

Listado 3-11. *Agregando un hermoso gradiente de fondo a nuestra caja.*

Los gradientes son configurados como fondos, por lo que podemos usar las propiedades **`background`** o **`background-image`** para declararlos. La sintaxis para los valores declarados en estas propiedades es **`linear-gradient(posición inicio, color inicial, color final)`**. Los atributos de la función **`linear-gradient()`** indican el punto de comienzo y los colores usados para crear el gradiente. El primer valor puede ser especificado en pixeles, porcentaje o usando las palabras clave **`top`**, **`bottom`**, **`left`** y **`right`** (como hicimos en nuestro ejemplo). El punto de comienzo puede ser reemplazado por un ángulo para declarar una dirección específica del gradiente:

```
background: linear-gradient(30deg, #FFFFFF, #006699);
```

Listado 3-12. *Gradiente con un ángulo de dirección de 30 grados.*

También podemos declarar los puntos de terminación para cada color:

```
background: linear-gradient(top, #FFFFFF 50%, #006699 90%);
```

Listado 3-13. *Declarando puntos de terminación.*

Gradiente radial

La sintaxis estándar para los gradientes radiales solo difiere en unos pocos aspectos con respecto a la anterior. Debemos usar la función **`radial-gradient()`** y un nuevo atributo para la forma:

```
background: radial-gradient(center, circle, #FFFFFF 0%, #006699 200%);
```

Listado 3-14. *Gradiente radial.*

La posición de comienzo es el origen y puede ser declarada en pixeles, porcentaje o una combinación de las palabras clave **`center`**, **`top`**, **`bottom`**, **`left`** y **`right`**. Existen dos posibles valores para la forma (**`circle`** y **`ellipse`**) y la terminación para el color indica el color y la posición donde las transiciones comienzan.

> **Hágalo usted mismo:** Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-14 para probar el efecto en su navegador (no olvide agregar los prefijos **`-moz-`** o **`-webkit-`** dependiendo del navegador que esté usando).

> **IMPORTANTE:** En este momento el efecto de gradientes ha sido implementado por los navegadores en diferentes formas. Lo que hemos aprendido en este capítulo es el estándar propuesto por W3C (World Wide Web Consortium). Navegadores como Firefox y Google Chrome ya incorporan una implementación que trabaja con este estándar, pero Internet Explorer y otros aún se encuentran ocupados en ello. Como siempre, pruebe sus códigos en cada navegador disponible en el mercado para comprobar el estado actual de las diferentes implementaciones.

RGBA

Hasta este momento los colores fueron declarados como sólidos utilizando valores hexadecimales o la función **`rgb()`** para decimales. CSS3 ha agregado una nueva función llamada **`rgba()`** que simplifica la asignación de colores y transparencias. Esta función además resuelve un problema previo provocado por la propiedad **`opacity`**.

La función **`rgba()`** tiene cuatro atributos. Los primeros tres son similares a los usados en **`rgb()`** y simplemente declaran los valores para los colores rojo, verde y azul en números decimales del 0 al 255. El último, en cambio, corresponde a la nueva capacidad de opacidad. Este valor se debe encontrar dentro de un rango que va de 0 a 1, con 0 como totalmente transparente y 1 como totalmente opaco.

```
#titulo {
  font: bold 36px MiNuevaFuente, verdana, sans-serif;
  text-shadow: rgba(0,0,0,0.5) 3px 3px 5px;
}
```

Listado 3-15. *Mejorando la sombra del texto con transparencia.*

El Listado 3-15 ofrece un simple ejemplo que demuestra cómo los efectos son mejorados aplicando transparencia. Reemplazamos la función **`rgb()`** por **`rgba()`** en la sombra del título y agregamos un valor de opacidad/transparencia de **`0.5`**. Ahora la sombra de nuestro título se mezclará con el fondo, creando un efecto mucho más natural.

En previas versiones de CSS teníamos que usar diferentes técnicas en diferentes navegadores para hacer un elemento transparente. Todas presentaban el mismo problema: el valor de opacidad de un elemento era heredado por sus hijos. Ese

problema fue resuelto por **`rgba()`** y ahora podemos asignar un valor de opacidad al fondo de una caja sin afectar su contenido.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-15 para probar el efecto en su navegador.

HSLA

Del mismo modo que la función **`rgba()`** agrega un valor de opacidad a **`rgb()`**, la función **`hsla()`** hace lo mismo para la función **`hsl()`**.

La función **`hsla()`** es simplemente un función diferente para generar colores, pero es más intuitiva que **`rgba()`**. Algunos diseñadores encontrarán más fácil generar un set de colores personalizado utilizando **`hsla()`**. La sintaxis de esta función es: **`hsla(tono, saturación, luminosidad, opacidad)`**.

```
#titulo {
  font: bold 36px MiNuevaFuente, verdana, sans-serif;
  text-shadow: rgba(0,0,0,0.5) 3px 3px 5px;
  color: hsla(120, 100%, 50%, 0.5);
}
```

Listado 3-16. *Nuevo color para el título usando `hsla()`.*

Siguiendo la sintaxis, **`tono`** representa el color extraído de una rueda imaginaria y es expresado en grados desde 0 a 360. Cerca de 0 y 360 están los colores rojos, cerca de 120 los verdes y cerca de 240 los azules. El valor **`saturación`** es representado en porcentaje, desde 0% (escala de grises) a 100% (todo color o completamente saturado). La **`luminosidad`** es también un valor en porcentaje desde 0% (completamente oscuro) a 100% (completamente iluminado). El valor 50% representa luminosidad normal o promedio. El último valor, así como en **`rgba()`**, representa la opacidad.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-16 para probar el efecto en su navegador.

Outline

La propiedad **`outline`** es una vieja propiedad CSS que ha sido expandida en CSS3 para incluir un valor de desplazamiento. Esta propiedad era usada para crear un segundo borde, y ahora ese borde puede ser mostrado alejado del borde real del elemento.

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  outline: 2px dashed #000099;
  outline-offset: 15px;
}
```

Listado 3-17. *Agregando un segundo borde a la cabecera.*

En el Listado 3-17 agregamos a los estilos originalmente aplicados a la caja de nuestra plantilla un segundo borde de 2 pixeles con un desplazamiento de 15 pixeles. La propiedad **`outline`** tiene similares características y usa los mismos parámetros que **`border`**. La propiedad **`outline-offset`** solo necesita un valor en pixeles.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-17 para probar el efecto en su navegador.

Border-image

Los posibles efectos logrados por las propiedades **`border`** y **`outline`** están limitados a líneas simples y solo algunas opciones de configuración. La nueva propiedad **`border-image`** fue incorporada para superar estas limitaciones y dejar en manos del diseñador la calidad y variedad de bordes disponibles ofreciendo la alternativa de utilizar imágenes propias.

> **Hágalo usted mismo:** Vamos a utilizar una imagen PNG que incluye diamantes para probar esta propiedad. Siga el siguiente enlace para descargar el archivo **`diamonds.png`** desde nuestro sitio web y luego copie este archivo en el mismo directorio (carpeta) donde se encuentra su archivo CSS: *www.minkbooks.com/ content/diamonds.png*.

La propiedad **`border-image`** toma una imagen y la utiliza como patrón. De acuerdo a los valores otorgados, la imagen es cortada como un pastel, las partes obtenidas son luego ubicadas alrededor del objeto para construir el borde.

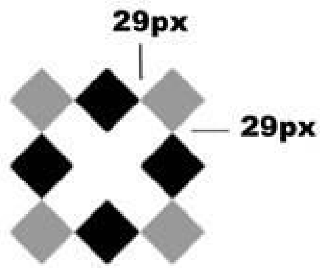


Figura 3-1. *Este es el patrón desde el cual vamos a construir nuestro borde. Cada pieza es de 29 pixeles de ancho, como indica la figura.*

Para hacer el trabajo, necesitamos especificar tres atributos: el nombre del archivo de la imagen, el tamaño de las piezas que queremos obtener del patrón y algunas palabras clave para declarar cómo las piezas serán distribuidas alrededor del objeto.

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 29px;
  -moz-border-image: url("diamonds.png") 29 stretch;
  -webkit-border-image: url("diamonds.png") 29 stretch;
  border-image: url("diamonds.png") 29 stretch;
}
```

Listado 3-18. *Un borde personalizado para la cabecera.*

Con las modificaciones realizadas en el Listado 3-18 estamos definiendo un borde de 29 pixeles para la caja de nuestra cabecera y luego cargando la imagen **`diamonds.png`** para construir ese borde. El valor 29 en la propiedad **`border-image`** declara el tamaño de las piezas y **`stretch`** es uno de los métodos disponibles para distribuir estas piezas alrededor de la caja.

Existen tres valores posibles para el último atributo. La palabra clave **`repeat`** repetirá las piezas tomadas de la imagen todas las veces que sea necesario para cubrir el lado del elemento. En este caso, el tamaño de las piezas es preservado y la imagen será cortada si no existe más espacio para ubicarla. La palabra clave **`round`** considerará qué tan largo es el lado a ser cubierto y ajustará el tamaño de las piezas para asegurarse que cubren todo el lado y ninguna pieza es cortada. Finalmente, la palabra clave **`stretch`** (usada en el Listado 3-18) estira solo una pieza para cubrir el lado completo.

En nuestro ejemplo utilizamos la propiedad **`border`** para definir el tamaño del borde, pero se puede también usar **`border-with`** para especificar diferentes tamaños para cada lado del elemento (la propiedad **`border-with`** usa cuatro parámetros, con una sintaxis similar a **`margin`** y **`padding`**). Lo mismo ocurre con el tamaño de cada pieza, hasta cuatro valores pueden ser declarados para obtener diferentes imágenes de diferentes tamaños desde el patrón.

> **Hágalo usted mismo:** Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-18 para probar el efecto en su navegador.

Transform y transition

Los elementos HTML, cuando son creados, son como bloques sólidos e inamovibles. Pueden ser movidos usando código Javascript o aprovechando librerías populares como jQuery (*www.jquery.com*), por ejemplo, pero no existía un procedimiento estándar para este propósito hasta que CSS3 presentó las propiedades **`transform`** y **`transition`**.

Ahora ya no tenemos que pensar en cómo hacerlo. En su lugar, solo tenemos que conocer cómo ajustar unos pocos parámetros y nuestro sitio web puede ser tan flexible y dinámico como lo imaginamos.

La propiedad **`transform`** puede operar cuatro transformaciones básicas en un elemento: **`scale`** (escalar), **`rotate`** (rotar), **`skew`** (inclinar) y **`translate`** (trasladar o mover). Veamos cómo funcionan:

Transform: scale

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: scale(2);
  -webkit-transform: scale(2);
}
```

Listado 3-19. *Cambiando la escala de la caja de la cabecera.*

En el ejemplo del Listado 3-19 partimos de los estilos básicos utilizados para la cabecera generada en el Listado 3-2 y aplicamos transformación duplicando la escala del elemento. La función **`scale`** recibe dos parámetros: el valor **`X`** para la escala horizontal y el valor **`Y`** para la escala vertical. Si solo un valor es provisto el mismo valor es aplicado a ambos parámetros.

Números enteros y decimales pueden ser declarados para la escala. Esta escala es calculada por medio de una matriz. Los valores entre 0 y 1 reducirán el elemento, un valor de 1 mantendrá las proporciones originales y valores mayores que 1 aumentarán las dimensiones del elemento de manera incremental.

Un efecto atractivo puede ser logrado con esta función otorgando valores negativos:

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: scale(1,-1);
  -webkit-transform: scale(1,-1);
}
```

Listado 3-20. *Creando una imagen espejo con `scale`.*

En el Listado 3-20, dos parámetros han sido declarados para cambiar la escala de la caja **`principal`**. El primer valor, 1, mantiene la proporción original para la dimensión horizontal de la caja. El segundo valor también mantiene la proporción original, pero invierte el elemento verticalmente para producir el efecto espejo.

Existen también otras dos funciones similares a **`scale`** pero restringidas a la dimensión horizontal o vertical: **`scaleX`** y **`scaleY`**. Estas funciones, por supuesto, utilizan un solo parámetro.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-19 o 3-20 para probar el efecto en su navegador.

Transform: rotate

La función **`rotate`** rota el elemento en la dirección de las agujas de un reloj. El valor debe ser especificado en grados usando la unidad “**`deg`**”:

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: rotate(30deg);
  -webkit-transform: rotate(30deg);
}
```

Listado 3-21. *Rotando la caja.*

Si un valor negativo es declarado, solo cambiará la dirección en la cual el elemento es rotado.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-21 para probar el efecto en su navegador.

Transform: skew

Esta función cambia la simetría del elemento en grados y en ambas dimensiones.

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: skew(20deg);
  -webkit-transform: skew(20deg);
}
```

Listado 3-22. *Inclinar horizontalmente.*

La función **`skew`** usa dos parámetros, pero a diferencia de otras funciones, cada parámetro de esta función solo afecta una dimensión (los parámetros actúan de forma independiente). En el Listado 3-22, realizamos una operación **`transform`** a la caja de la cabecera para inclinarla. Solo declaramos el primer parámetro, por lo que solo la dimensión horizontal de la caja será modificada. Si usáramos los dos parámetros, podríamos alterar ambas dimensiones del objeto. Como alternativa podemos utilizar funciones diferentes para cada una de ellas: **`skewX`** y **`skewY`**.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-22 para probar el efecto en su navegador.

Transform: translate

Similar a las viejas propiedades **`top`** y **`left`**, la función **`translate`** mueve o desplaza el elemento en la pantalla a una nueva posición.

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: translate(100px);
  -webkit-transform: translate(100px);
}
```

Listado 3-23. *Moviendo la caja de la cabecera hacia la derecha.*

La función **`translate`** considera la pantalla como una grilla de pixeles, con la posición original del elemento usada como un punto de referencia. La esquina superior izquierda del elemento es la posición **`0,0`**, por lo que valores negativos moverán al objeto hacia la izquierda o hacia arriba de la posición original, y valores positivos lo harán hacia la derecha o hacia abajo.

En el Listado 3-23, movimos la caja de la cabecera hacia la derecha unos 100 pixeles desde su posición original. Dos valores pueden ser declarados en esta función si queremos mover el elemento horizontal y verticalmente, o podemos usar funciones independientes llamadas **`translateX`** y **`translateY`**.

> **Hágalo usted mismo:** Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-23 para probar el efecto en su navegador.

Transformando todo al mismo tiempo

A veces podría resultar útil realizar sobre un elemento varias transformaciones al mismo tiempo. Para obtener una propiedad **`transform`** combinada, solo tenemos que separar cada función a aplicar con un espacio:

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: translateY(100px) rotate(45deg) scaleX(0.3);
  -webkit-transform: translateY(100px) rotate(45deg) scaleX(0.3);
}
```

Listado 3-24. *Moviendo, escalando y rotando el elemento con solo una línea de código.*

Una de las cosas que debe recordar en este caso es que el orden es importante. Esto es debido a que algunas funciones mueven el punto original y el centro del objeto, cambiando de este modo los parámetros que el resto de las funciones utilizarán para operar.

> **Hágalo usted mismo:** Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-24 para probar el efecto en su navegador.

Transformaciones dinámicas

Lo que hemos aprendido hasta el momento en este capítulo cambiará la forma de la web, pero la mantendrá tan estática como siempre. Sin embargo, podemos aprovecharnos de la combinación de transformaciones y pseudo clases para convertir nuestra página en una aplicación dinámica:

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
}
#principal:hover{
  -moz-transform: rotate(5deg);
  -webkit-transform: rotate(5deg);
}
```

Listado 3-25. *Respondiendo a la actividad del usuario.*

En el Listado 3-25, la regla original del Listado 3-2 para la caja de la cabecera fue conservada intacta, pero una nueva regla fue agregada para aplicar efectos de transformación usando la vieja pseudo clase **`:hover`**. El resultado obtenido es que cada vez que el puntero del ratón pasa sobre esta caja, la propiedad **`transform`** rota la caja en 5 grados, y cuando el puntero se aleja la caja vuelve a rotar de regreso a su posición original. Este efecto produce una animación básica pero útil con nada más que propiedades CSS.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-25 para probar el efecto en su navegador.

Transiciones

De ahora en más, hermosos efectos usando transformaciones dinámicas son accesibles y fáciles de implementar. Sin embargo, una animación real requiere de un proceso de más de dos pasos.

La propiedad **`transition`** fue incluida para suavizar los cambios, creando mágicamente el resto de los pasos que se encuentran implícitos en el movimiento. Solo agregando esta propiedad forzamos al navegador a tomar cartas en el asunto, crear para nosotros todos esos pasos invisibles, y generar una transición suave desde un estado al otro.

```
#principal {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transition: -moz-transform 1s ease-in-out 0.5s;
  -webkit-transition: -webkit-transform 1s ease-in-out 0.5s;
}
#principal:hover{
  -moz-transform: rotate(5deg);
  -webkit-transform: rotate(5deg);
}
```

Listado 3-26. *Una hermosa rotación usando transiciones.*

Como puede ver en el Listado 3-26, la propiedad **`transition`** puede tomar hasta cuatro parámetros separados por un espacio. El primer valor es la propiedad que será considerada para hacer la transición (en nuestro ejemplo elegimos **`transform`**). Esto es necesario debido a que varias propiedades pueden cambiar al mismo tiempo y probablemente necesitemos crear los pasos del proceso de transición solo para una de ellas. El segundo parámetro especifica el tiempo que la transición se tomará para ir de la posición inicial a la final. El tercer parámetro puede ser cualquiera de las siguientes palabras clave: **`ease`**, **`linear`**, **`ease-in`**, **`ease-out`** o **`ease-in-out`**. Estas palabras clave determinan cómo se realizará el proceso de transición basado en una curva Bézier. Cada una de ellas representa diferentes tipos de curva Bézier, y la mejor forma de saber cómo trabajan es viéndolas funcionar en pantalla. El último parámetro para la propiedad **`transition`** es el retardo. Éste indica cuánto tiempo tardará la transición en comenzar.

Para producir una transición para todas las propiedades que están cambiando en un objeto, la palabra clave **`all`** debe ser especificada. También podemos declarar varias propiedades a la vez listándolas separadas por coma.

Hágalo usted mismo: Reemplace el correspondiente código del Listado 3-11 por el código del Listado 3-26 para probar el efecto en su navegador.

IMPORTANTE: En el Listado 3-26 realizamos una transición con la propiedad **`transform`**. No todas las propiedades CSS son soportadas por la propiedad **`transition`** en este momento y probablemente la lista cambie con el tiempo. Deberá probar cada una de ellas por usted mismo o visitar el sitio web oficial de cada navegador para encontrar más información al respecto.

3.2 Referencia rápida

CSS3 provee nuevas propiedades para crear efectos visuales y dinámicos que son parte esencial de la web en estos días.

border-radius Esta propiedad genera esquinas redondeadas para la caja formada por el elemento. Posee dos parámetros diferentes que dan forma a la esquina. El primer parámetro determina la curvatura horizontal y el segundo la vertical, otorgando la posibilidad de crear una elipsis. Para declarar ambos parámetros de la curva, los valores deben ser separados por una barra (por ejemplo, **`border-radius: 15px / 20px`**). Usando solo un valor determinaremos la misma forma para todas las esquinas (por ejemplo, **`border-radius: 20px`**). Un valor para cada esquina puede ser declarado en un orden que sigue las agujas del reloj, comenzando por la esquina superior izquierda.

box-shadow Esta propiedad crea sombras para la caja formada por el elemento. Puede tomar cinco parámetros: el color, el desplazamiento horizontal, el desplazamiento vertical, el valor de difuminación, y la palabra clave **`inset`** para generar una sombra interna. Los desplazamientos pueden ser negativos, y el valor de difuminación y el valor **`inset`** son opcionales (por ejemplo, **`box-shadow: #000000 5px 5px 10px inset`**).

text-shadow Esta propiedad es similar a **`box-shadow`** pero específica para textos. Toma cuatro parámetros: el color, el desplazamiento horizontal, el desplazamiento vertical, y el valor de difuminación (por ejemplo, **`text-shadow: #000000 5px 5px 10px`**).

@font-face Esta regla nos permite cargar y usar cualquier fuente que necesitemos. Primero, debemos declarar la fuente, proveer un nombre con la propiedad **`font-family`** y especificar el archivo con **`src`** (por ejemplo, **`@font-face{ font-family: Mifuente; src: url('font.ttf') }`**). Luego de esto, podremos asignar la fuente (en el ejemplo **`Mifuente`**) a cualquier elemento del documento.

linear-gradient(posición inicio, color inicial, color final) Esta función puede ser aplicada a las propiedades **`background`** o **`background-image`** para generar un gradiente lineal. Los atributos indican el punto inicial y los colores usados para crear el gradiente. El primer valor puede ser especificado en pixeles, en porcentaje o usando las palabras clave **`top`**, **`bottom`**, **`left`** y **`right`**. El punto de inicio puede ser reemplazado por un ángulo para proveer una dirección específica para el gradiente (por ejemplo, **`linear-gradient(top, #FFFFFF 50%, #006699 90%);`**).

radial-gradient(posición inicio, forma, color inicial, color final) Esta función puede ser aplicada a las propiedades **`background`** o **`background-image`** para generar un gradiente radial. La posición de inicio es el origen y puede ser declarado en pixeles, porcentaje o como una combinación de las palabras clave **`center`**, **`top`**, **`bottom`**, **`left`** y **`right`**. Existen dos valores para la forma: **`circle`** y **`ellipse`**, y puntos de terminación pueden ser declarados para cada color indicando la posición donde la transición comienza (por ejemplo, **`radial-gradient(center, circle, #FFFFFF 0%, #006699 200%);`**).

rgba() Esta función es una mejora de **`rgb()`**. Toma cuatro valores: el color rojo (0-255), el color verde (0-255), el color azul (0-255), y la opacidad (un valor entre 0 y 1).

hsla() Esta función es una mejora de **`hsl()`**. Puede tomar cuatro valores: el tono (un valor entre 0 y 360), la saturación (un porcentaje), la luminosidad (un porcentaje), y la opacidad (un valor entre 0 y 1).

outline Esta propiedad fue mejorada con la incorporación de otra propiedad llamada **`outline-offset`**. Ambas propiedades combinadas generan un segundo borde alejado del borde original del elemento (por ejemplo, **`outline: 1px solid #000000; outline-offset: 10px;`**).

border-image Esta propiedad crea un borde con una imagen personalizada. Necesita que el borde sea declarado previamente con las propiedades **`border`** o **`border-with`**, y toma al menos tres parámetros: la URL de la imagen, el tamaño de las piezas que serán tomadas de la imagen para construir el borde, y una palabra clave que especifica cómo esas piezas serán ubicadas alrededor del elemento (por ejemplo, **`border-image: url("file.png") 15 stretch;`**).

transform Esta propiedad modifica la forma de un elemento. Utiliza cuatro funciones básicas: **`scale`** (escalar), **`rotate`** (rotar), **`skew`** (inclinar), y **`translate`** (trasladar o mover). La función **`scale`** recibe solo un parámetro. Un valor negativo invierte el elemento, valores entre 0 y 1 reducen el elemento y valores mayores que 1 expanden el elemento (por ejemplo, **`transform: scale(1.5);`**). La función **`rotate`** usa solo un parámetro expresado en grados para rotar el elemento (por ejemplo, **`transform: rotate(20deg);`**). La función **`skew`** recibe dos valores, también en grados, para la transformación horizontal y vertical (por ejemplo, **`transform: skew(20deg, 20deg);`**). La función **`translate`** mueve el objeto tantos pixeles como sean especificados por sus parámetros (por ejemplo, **`transform: translate(20px);`**).

transition Esta propiedad puede ser aplicada para crear una transición entre dos estados de un elemento. Recibe hasta cuatro parámetros: la propiedad afectada, el tiempo que le tomará a la transición desde el comienzo hasta el final, una palabra clave para especificar cómo la transición será realizada (**`ease`**, **`linear`**, **`ease-in`**, **`ease-out`**,

ease-in-out) y un valor de retardo que determina el tiempo que la transición tardará en comenzar (por ejemplo, **transition: color 2s linear 1s;**).

Capítulo 4
Javascript

4.1 La relevancia de Javascript

HTML5 puede ser imaginado como un edificio soportado por tres grandes columnas: HTML, CSS y Javascript. Ya hemos estudiado los elementos incorporados en HTML y las nuevas propiedades que hacen CSS la herramienta ideal para diseñadores. Ahora es momento de develar lo que puede ser considerado como uno de los pilares más fuertes de esta especificación: Javascript.

Javascript es un lenguaje interpretado usado para múltiples propósitos pero solo considerado como un complemento hasta ahora. Una de las innovaciones que ayudó a cambiar el modo en que vemos Javascript fue el desarrollo de nuevos motores de interpretación, creados para acelerar el procesamiento de código. La clave de los motores más exitosos fue transformar el código Javascript en código máquina para lograr velocidades de ejecución similares a aquellas encontradas en aplicaciones de escritorio. Esta mejorada capacidad permitió superar viejas limitaciones de rendimiento y confirmar el lenguaje Javascript como la mejor opción para la web.

Para aprovechar esta prometedora plataforma de trabajo ofrecida por los nuevos navegadores, Javascript fue expandido en relación con portabilidad e integración. A la vez, interfaces de programación de aplicaciones (APIs) fueron incorporadas por defecto en cada navegador para asistir al lenguaje en funciones elementales. Estas nuevas APIs (como Web Storage, Canvas, y otras) son interfaces para librerías incluidas en navegadores. La idea es hacer disponible poderosas funciones a través de técnicas de programación sencillas y estándares, expandiendo el alcance del lenguaje y facilitando la creación de programas útiles para la web.

En este capítulo vamos a estudiar cómo incorporar Javascript dentro de nuestros documentos HTML y veremos las incorporaciones recientes a este lenguaje con la intención de prepararnos para el resto del libro.

> **IMPORTANTE:** Nuestro acercamiento a Javascript en este libro es introductorio. Al igual que con CSS, solo vamos a repasar las técnicas que necesitamos entender para implementar los ejemplos de próximos capítulos. Trabajaremos sobre temas complejos, pero solo usando un mínimo de código necesario para aprovechar estas nuevas características. Si necesita expandir sus conocimientos sobre este lenguaje, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

4.2 Incorporando Javascript

Siguiendo los mismos lineamientos que en CSS, existen tres técnicas para incorporar código Javascript dentro de HTML. Sin embargo, al igual que en CSS, solo la inclusión de archivos externos es la recomendada a usar en HTML5.

> **IMPORTANTE:** En este capítulo introducimos nuevas características así como técnicas básicas necesarias para entender los ejemplos del libro. Si usted se encuentra familiarizado con esta información siéntase libre de obviar las partes que ya conoce.

En línea

Esta es una técnica simple para insertar Javascript en nuestro documento que se aprovecha de atributos disponibles en elementos HTML. Estos atributos son manejadores de eventos que ejecutan código de acuerdo a la acción del usuario.

Los manejadores de eventos más usados son, en general, los relacionados con el ratón, como por ejemplo **onclick**, **onMouseOver**, u **onMouseOut**. Sin embargo, encontraremos sitios web que implementan eventos de teclado y de la ventana, ejecutando acciones luego de que una tecla es presionada o alguna condición en la ventana del navegador cambia (por ejemplo, **onload** u **onfocus**).

```
<!DOCTYPE html>
<html lang=”es”>
<head>
  <title>Este texto es el título del documento</title>
</head>
<body>
  <div id=”principal”>
    <p onclick=”alert(‘hizo clic!’)”>Hacer Clic</p>
    <p>No puede hacer clic</p>
  </div>
</body>
</html>
```

Listado 4-1. Javascript *en línea.*

Usando el manejador de eventos **onclick** en el Listado 4-1, un código es ejecutado cada vez que el usuario hace clic con el ratón sobre el texto que dice “Hacer Clic”. Lo que el manejador **onclick** está diciendo es algo como: “cuando alguien haga clic sobre este elemento ejecute este código” y el código en este caso es una función predefinida en Javascript que muestra una pequeña ventana con el mensaje “hizo clic!”.

Intente cambiar el manejador **onclick** por **onMouseOver**, por ejemplo, y verá cómo el código es ejecutado solo pasando el puntero del ratón sobre el elemento.

El uso de Javascript dentro de etiquetas HTML está permitido en HTML5, pero por las mismas razones que en CSS, esta clase de práctica no es recomendable. El código HTML se extiende innecesariamente y se hace difícil de mantener y actualizar. Así mismo, el código distribuido sobre todo el documento complica la construcción de aplicaciones útiles.

Nuevos métodos y técnicas fueron desarrollados para referenciar elementos HTML y registrar manejadores de eventos sin tener que usar código en línea (*inline*). Volveremos sobre esto y aprenderemos más sobre eventos y manejadores de eventos más adelante en este capítulo.

> **Hágalo usted mismo:** Copie el código del Listado 4-1 y los siguientes códigos estudiados en este capítulo dentro de un nuevo archivo HTML. Abra el archivo en su navegador para probar cada ejemplo.

Embebido

Para trabajar con códigos extensos y funciones personalizadas debemos agrupar los códigos en un mismo lugar entre etiquetas **<script>**. El elemento **<script>** actúa exactamente igual al elemento **<style>** usado para incorporar estilos CSS. Nos ayuda a organizar el código en un solo lugar, afectando a los elementos HTML por medio de referencias.

Del mismo modo que con el elemento **<style>**, en HTML5 no debemos usar ningún atributo para especificar lenguaje. Ya no es necesario incluir el atributo **type** en la etiqueta **<script>**. HTML5 asigna Javascript por defecto.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <script>
    function mostraralerta(){
      alert('hizo clic!');
    }
    function hacerclic(){
      document.getElementsByTagName('p')[0].onclick=mostraralerta;
    }
    window.onload=hacerclic;
  </script>
</head>
<body>
  <div id="principal">
    <p>Hacer Clic</p>
    <p>No puede hacer Clic</p>
  </div>
</body>
</html>
```

Listado 4-2. Javascript embebido.

El elemento **<script>** y su contenido pueden ser posicionados en cualquier lugar del documento, dentro de otros elementos o entre ellos. Para mayor claridad, recomendamos siempre colocar sus códigos Javascript en la cabecera del documento (como en el ejemplo del Listado 4-2) y luego referenciar los elementos a ser afectados usando los métodos Javascript apropiados para ese propósito.

Actualmente existen tres métodos disponibles para referenciar elementos HTML desde Javascript:

- **getElementsByTagName** (usado en el Listado 4-2) referencia un elemento por su nombre o palabra clave.
- **getElementById** referencia un elemento por el valor de su atributo **id**.
- **getElementsByClassName** es una nueva incorporación que nos permite referenciar un elemento por el valor de su atributo **class**.

Incluso si seguimos la práctica recomendada (posicionar el código dentro de la cabecera del documento), una situación debe ser considerada: el código del documento es leído de forma secuencial por el navegador y no podemos referenciar un elemento que aún no ha sido creado.

En el Listado 4-2, el código es posicionado en la cabecera del documento y es leído por el navegador previo a la creación del elemento **<p>** que estamos referenciando. Si hubiésemos intentado afectar el elemento **<p>** directamente con una referencia, hubiéramos recibido un mensaje de error anunciando que el elemento no existe. Para evitar este problema, el código fue convertido a una función llamada **mostraralerta()**, y la referencia al elemento **<p>** junto con el manejador del evento fueron colocados en una segunda función llamada **hacerclic()**.

Las funciones son llamadas desde la última línea del código usando otro manejador de eventos (en este caso asociado con la ventana) llamado **onload**. Este manejador ejecutará la función **hacerclic()** cuando el documento sea completamente cargado y todos los elementos creados.

Es tiempo de analizar cómo el documento del Listado 4-2 es ejecutado. Primero las funciones Javascript son cargadas (declaradas) pero no ejecutadas. Luego los elementos HTML, incluidos los elementos **<p>**, son creados. Y finalmente, cuando el documento completo es cargado en la ventana del navegador, el evento **load** es disparado y la función **hacerclic()** es llamada.

En esta función, el método **getElementsByTagName** referencia todos los elementos **<p>**. Este método retorna un arreglo (array) conteniendo una lista de los elementos de la clase especificada encontrados en el documento. Sin embargo, usando el índice **[0]** al final del método indicamos que solo queremos que el primer elemento de la lista sea retornado. Una vez que este elemento es identificado, el código registra el manejador de eventos **onclick** para el mismo. La función **mostraralerta()** será ejecutada cuando el evento **click** es disparado sobre este elemento mostrando el mensaje “hizo clic!”.

Puede parecer mucho código y trabajo para reproducir el mismo efecto logrado por una simple línea en el ejemplo del Listado 4-1. Sin embargo, considerando el potencial de HTML5 y la complejidad alcanzada por Javascript, la concentración del código en un único lugar y la apropiada organización representa una gran ventaja para nuestras futuras

implementaciones y para hacer nuestros sitios web y aplicaciones fáciles de desarrollar y mantener.

Conceptos básicos: Una función es un código agrupado que es ejecutado solo cuando la función es invocada (activada o llamada) por su nombre. Normalmente, una función es llamada usando su nombre y algunos valores encerrados entre paréntesis (por ejemplo, **`hacerclic(1,2)`**). Una excepción a esta sintaxis es usada en el Listado 4-2. En este código no usamos paréntesis porque estamos pasando al evento solo la referencia a la función, no el resultado de su ejecución. Para aprender más sobre funciones Javascript, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Archivos externos

Los códigos Javascript crecen exponencialmente cuando agregamos nuevas funciones y aplicamos algunas de las APIs mencionadas previamente. Códigos embebidos incrementan el tamaño de nuestros documentos y los hacen repetitivos (cada documento debe volver a incluir los mismos códigos). Para reducir los tiempos de descarga, incrementar nuestra productividad y poder distribuir y reusar nuestros códigos en cada documento sin comprometer eficiencia, recomendamos grabar todos los códigos Javascript en uno o más archivos externos y llamarlos usando el atributo **`src`**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <script src=”micodigo.js”></script>
</head>
<body>
  <div id=”principal”>
    <p>Hacer Clic</p>
    <p>No puede hacer Clic</p>
  </div>
</body>
</html>
```

Listado 4-3. *Cargando el código desde archivos externos.*

El elemento **`<script>`** en el Listado 4-3 carga los códigos Javascript desde un archivo externo llamado **`micodigo.js`**. De ahora en más, podremos insertar nuestros códigos en este archivo y luego incluir el mismo en cualquier documento de nuestro sitio web que lo necesite. Desde la perspectiva del usuario, esta práctica reduce tiempos de descarga y acceso a nuestro sitio web, mientras que para nosotros simplifica la organización y facilita el mantenimiento.

Hágalo usted mismo: Copie el código del Listado 4-3 dentro del archivo HTML previamente creado. Cree un nuevo archivo de texto vacío llamado **`micodigo.js`** y copie en su interior el código Javascript del Listado 4-2. Tenga en cuenta que solo el código entre las etiquetas **`<script>`** debe ser copiado, sin incluir las etiquetas mismas.

4.3 Nuevos Selectores

Como vimos anteriormente, los elementos HTML tienen que ser referenciados desde Javascript para ser afectados por el código. Si recuerda de previos capítulos, CSS, y especialmente CSS3, ofrece un poderoso sistema de referencia y selección que no tiene comparación con los pocos métodos provistos por Javascript para este propósito. Los métodos **getElementById**, **getElementsByTagName** y **getElementsByClassName** no son suficientes para contribuir a la integración que este lenguaje necesita y sostener la relevancia que posee dentro de la especificación de HTML5. Para elevar Javascript al nivel que las circunstancias requieren, nuevas alternativas debieron ser incorporadas. Desde ahora podemos seleccionar elementos HTML aplicando toda clase de selectores CSS por medio de los nuevos métodos **querySelector()** y **querySelectorAll()**.

querySelector()

Este método retorna el primer elemento que concuerda con el grupo de selectores especificados entre paréntesis. Los selectores son declarados usando comillas y la misma sintaxis CSS, como en el siguiente ejemplo:

```
function hacerclic(){
  document.querySelector("#principal p:first-
                                  child").onclick=mostraralerta;
}
function mostraralerta(){
  alert('hizo clic!');
}
window.onload=hacerclic;
```

Listado 4-4. Usando `querySelector()`.

En el Listado 4-4, el método **getElementsByTagName** usado anteriormente ha sido reemplazado por **querySelector()**. Los selectores para esta consulta en particular están referenciando al primer elemento **<p>** que es hijo del elemento identificado con el atributo **id** y el valor **main**.

Debido a que ya explicamos que este método solo retorna el primer elemento encontrado, probablemente notará que la pseudo clase **first-child** es redundante. El método **querySelector()** en nuestro ejemplo retornará el primer elemento **<p>** dentro de **<div>** que es, por supuesto, su primer hijo. El propósito de este ejemplo es mostrarle que **querySelector()** acepta toda clase de selectores válidos CSS y ahora, del mismo modo que en CSS, Javascript también provee herramientas importantes para referenciar cada elemento en el documento.

Varios grupos de selectores pueden ser declarados separados por coma. El método **querySelector()** retornará el primer elemento que concuerde con cualquiera de ellos.

> **Hágalo usted mismo:** Reemplace el código en el archivo **micodigo.js** por el provisto en el Listado 4-4 y abra el archivo HTML con el código del Listado 4-3 en su navegador para ver el método **querySelector()** en acción.

querySelectorAll()

En lugar de uno, el método **querySelectorAll()** retorna todos los elementos que concuerdan con el grupo de selectores declarados entre paréntesis. El valor retornado es un arreglo (array) conteniendo cada elemento encontrado en el orden en el que aparecen en el documento.

```
function hacerclic(){
  var lista=document.querySelectorAll("#principal p");
  lista[0].onclick=mostraralerta;
}
function mostraralerta(){
  alert('hizo clic!');
}
window.onload=hacerclic;
```

Listado 4-5. *Usando* `querySelectorAll()`.

El grupo de selectores especificados en el método **querySelectorAll()** del Listado 4-5 encontrará cada elemento **<p>** en el documento HTML del listado 4-3 que es hijo del elemento **<div>**. Luego de la ejecución de esta primera línea, el array **lista** tendrá dos valores: una referencia al primer elemento **<p>** y una referencia al segundo elemento **<p>**. Debido a que el índice de cada array comienza por 0, en la próxima línea el primer elemento encontrado es referenciado usando corchetes y el valor 0 (**lista[0]**).

Note que este ejemplo no muestra el potencial de **querySelectorAll()**. Normalmente será utilizado para afectar a varios elementos y no solo uno, como en este caso. Para interactuar con una lista de elementos retornados por este método, podemos utilizar un bucle **for**:

```
function hacerclic(){
  var lista=document.querySelectorAll("#principal p");
  for(var f=0; f<lista.length; f++){
    lista[f].onclick=mostraralerta;
  }
}
function mostraralerta(){
  alert('hizo clic!');
}
window.onload=hacerclic;
```

Listado 4-6. *Afectando todos los elementos encontrados por* `querySelectorAll()`.

En el Listado 4-6, en lugar de seleccionar solo el primer elemento encontrado, registramos el manejador de eventos **onclick** para cada uno de ellos usando un bucle **for**. Ahora, todos los elementos **<p>** dentro de **<div>** mostrarán una pequeña ventana cuando el usuario haga clic sobre ellos.

El método **querySelectorAll()**, al igual que **querySelector()**, puede contener uno o más grupos de selectores separados por coma. Éstos y los demás métodos estudiados pueden ser combinados para referenciar elementos a los que resulta difícil llegar. Por ejemplo, en el próximo listado, el mismo resultado del código del Listado 4-6 es logrado utilizando **querySelectorAll()** y **getElementById()** juntos:

```
function hacerclic(){
  var lista=document.getElementById('principal').
                                          querySelectorAll("p");
  lista[0].onclick=mostraralerta;
}
function mostraralerta(){
  alert('hizo clic!');
}
window.onload=hacerclic;
```

Listado 4-7. Combinando métodos.

Usando esta técnica podemos ver qué precisos pueden ser estos métodos. Podemos combinarlos en una misma línea y luego realizar una segunda selección con otro método para alcanzar elementos dentro de los primeros. En próximos capítulos estudiaremos algunos ejemplos más.

4.4 Manejadores de eventos

Como comentamos anteriormente, el código Javascript es normalmente ejecutado luego de que el usuario realiza alguna acción. Estas acciones y otros eventos son procesados por manejadores de eventos y funciones Javascript asociadas con ellos.

Existen tres diferentes formas de registrar un evento para un elemento HTML: podemos agregar un nuevo atributo al elemento, registrar un manejador de evento como una propiedad del elemento o usar el nuevo método estándar **`addEventListener()`**.

> **Conceptos básicos:** En Javascript las acciones de los usuarios son llamadas eventos. Cuando el usuario realiza una acción, como un clic del ratón o la presión de una tecla, un evento específico para cada acción y cada elemento es disparado. Además de los eventos producidos por los usuarios existen también otros eventos disparados por el sistema (por ejemplo, el evento **`load`** que se dispara cuando el documento es completamente cargado). Estos eventos son manejados por códigos o funciones. El código que responde al evento es llamado manejador. Cuando registramos un manejador lo que hacemos es definir cómo nuestra aplicación responderá a un evento en particular. Luego de la estandarización del método **`addEventListener()`**, este procedimiento es usualmente llamado "escuchar al evento", y lo que hacemos para preparar el código que responderá a ese evento es "agregar una escucha" a un elemento en particular.

Manejadores de eventos en línea

Ya utilizamos esta técnica en el código del Listado 4-1 incluyendo el atributo **`onclick`** en el elemento **`<p>`** (revise este código para comprobar cómo se aplica). Se trata simplemente de utilizar los atributos provistos por HTML para registrar eventos para un elemento en particular. Esta es una técnica en desuso pero aun extremadamente útil y práctica en algunas circunstancias, especialmente cuando necesitamos hacer modificaciones rápidas para testeo.

Manejadores de eventos como propiedades

Para evitar las complicaciones de la técnica en línea (inline), debemos registrar los eventos desde el código Javascript. Usando selectores Javascript podemos referenciar el elemento HTML y asignarle el manejador de eventos que queremos como si fuese una propiedad.

En el código del Listado 4-2 encontrará esta técnica puesta en práctica. Dos manejadores de eventos fueron asignados como propiedades a diferentes elementos. El manejador de eventos **`onload`** fue registrado para la ventana usando la construcción **`window.onload`**, y el manejador de eventos **`onclick`** fue registrado para el primer elemento **`<p>`** encontrado en el documento con la línea de código **`document.getElementsByTagName('p')[0].onclick.`**

> **Conceptos Básicos:** Los nombres de los manejadores de eventos son construidos agregando el prefijo **`on`** al nombre del evento. Por ejemplo, el nombre del manejador de eventos para el evento **`click`** es **`onclick`**. Cuando estamos hablando sobre **`onclick`** usualmente hacemos referencia al código que será ejecutado cuando el evento **`click`** se produzca.

Antes de HTML5, esta era la única técnica disponible para usar manejadores de eventos desde Javascript que funcionaba en todos los navegadores. Algunos fabricantes de navegadores estaban desarrollando sus propios sistemas, pero nada fue adoptado por todos hasta que el nuevo estándar fue declarado. Por este motivo recomendamos utilizar esta técnica por razones de compatibilidad, pero no la sugerimos para sus aplicaciones HTML5.

El método addEventListener()

El método **`addEventListener()`** es la técnica ideal y la que es considerada como estándar por la especificación de HTML5. Este método tiene tres argumentos: el nombre del evento, la función a ser ejecutada y un valor booleano (falso o verdadero) que indica cómo un evento será disparado en elementos superpuestos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Este texto es el título del documento</title>
  <script>
```

```
      function mostraralerta(){
        alert('hizo clic!');
      }
      function hacerclic(){
        var elemento=document.getElementsByTagName('p')[0];
        elemento.addEventListener('click', mostraralerta, false);
      }
      window.addEventListener('load', hacerclic, false);
    </script>
  </head>
  <body>
    <div id="principal">
      <p>Hacer Clic</p>
      <p>No puede hacer Clic</p>
    </div>
  </body>
  </html>
```

Listado 4-8. *Agregando escuchas para eventos con* `addEventListener()`.

El Listado 4-8 presenta el mismo código que el Listado 4-2 pero ahora una escucha fue agregada para cada evento usando el método **`addEventListener()`**. Para organizar el código en la función **`hacerclic()`**, asignamos la referencia al elemento a una variable llamada **`elemento`** y luego agregamos la escucha para el evento **`click`** usando esa variable.

La sintaxis del método **`addEventListener()`** es la mostrada en el Listado 4-8. El primer atributo es el nombre del evento. El segundo es la función a ser ejecutada, la cual puede ser una referencia a una función (como en este caso) o una función anónima. El tercer atributo especificará, usando **`true`** (verdadero) o **`false`** (falso), cómo múltiples eventos serán disparados. Por ejemplo, si estamos escuchando al evento **`click`** en dos elementos que se encuentran anidados (uno dentro de otro), cuando el usuario hace clic sobre estos elementos dos eventos **`click`** son disparados en un orden que depende de este valor. Si el atributo es declarado como **`true`** para uno de los elementos, entonces ese evento será considerado primero y el otro luego. Normalmente el valor **`false`** es el más adecuado para la mayoría de las situaciones.

> **Conceptos básicos:** Funciones anónimas son funciones dinámicamente declaradas y que no tienen nombre (por esto son llamadas "anónimas"). Esta clase de funciones son extremadamente útiles en Javascript, nos ayudan a organizar el código y no sobre poblar el objeto global con funciones independientes. Usaremos funciones anónimas con frecuencia en los siguientes capítulos.

Incluso cuando los resultados de aplicar esta técnica y la anterior son similares, **`addEventListener()`** nos permite agregar tantas escuchas como necesitemos para el mismo elemento. Esta distinción le otorga a **`addEventListener()`** una ventaja sobre el resto, convirtiéndola en la técnica ideal para aplicaciones HTML5.

Debido a que los eventos son la clave para sitios webs y aplicaciones interactivas, varios fueron agregados en la especificación de HTML5. En próximos capítulos estudiaremos cada uno de estos nuevos eventos y el entorno en el cual trabajan.

4.5 APIs

Si cuenta con alguna experiencia anterior en programación, o simplemente siguió este capítulo desde el comienzo, le resultará fácil reconocer la cantidad de código necesario para realizar tareas sencillas. Ahora imagine todo el trabajo que debería hacer para construir un sistema de bases de datos desde cero, o generar gráficos complejos en la pantalla, o crear una aplicación para manipulación de imágenes, por nombrar unos pocos.

Javascript es tan poderoso como cualquier otro lenguaje de desarrollo en este momento. Y por la misma razón que lenguajes de programación profesionales poseen librerías para crear elementos gráficos, motores 3D para video juegos o interfaces para acceder a bases de datos, Javascript cuenta con APIs para ayudar a los programadores a lidiar con actividades complejas.

HTML5 introduce varias APIs (interfaces de programación de aplicaciones) para proveer acceso a poderosas librerías desde simple código Javascript. El potencial de estas incorporaciones es tan importante que pronto se convertirán en nuestro objeto de estudio. Veamos rápidamente sus características para obtener una perspectiva de lo que nos encontraremos en el resto del libro.

La siguiente es solo una introducción, más adelante estudiaremos cada una de estas tecnologías con mayor profundidad.

Canvas

Canvas es una API gráfica que provee una básica pero poderosa superficie de dibujo. Esta es la más maravillosa y prometedora API de todas. La posibilidad de generar e imprimir gráficos en pantalla, crear animaciones o manipular imágenes y videos (combinado con la funcionalidad restante de HTML5) abre las puertas para lo que nos podamos imaginar.

Canvas genera una imagen con pixeles que son creados y manipulados por funciones y métodos provistos específicamente para este propósito.

Drag and Drop

Drag and Drop incorpora la posibilidad de arrastrar y soltar elementos en la pantalla como lo haríamos comúnmente en aplicaciones de escritorio. Ahora, con unas pocas líneas de código, podemos hacer que un elemento esté disponible para ser arrastrado y soltado dentro de otro elemento en la pantalla. Estos elementos pueden incluir no solo gráficos sino además textos, enlaces, archivos o datos en general.

Geolocation

Geolocation es utilizada para establecer la ubicación física del dispositivo usado para acceder a la aplicación. Existen varios métodos para acceder a esta información, desde señales de red hasta el Sistema de Posicionamiento Global (GPS). Los valores retornados incluyen latitud y longitud, posibilitando la integración de esta API con otras como Google Maps, por ejemplo, o acceder a información de localización específica para la construcción de aplicaciones prácticas que trabajen en tiempo real.

Storage

Dos APIs fueron creadas con propósitos de almacenamiento de datos: **Web Storage** e **Indexed Database**. Básicamente, estas APIs transfieren la responsabilidad por el almacenamiento de datos del servidor al ordenador del usuario, pero en el caso de Web Storage y su atributo sessionStorage, esta incorporación también incrementa el nivel de control y la eficiencia de las aplicaciones web.

Web Storage contiene dos importantes atributos que son a veces considerados APIs por sí mismos: sessionStorage y localStorage.

El atributo sessionStorage es responsable por mantener consistencia sobre la duración de la sesión de una página web y preservar información temporal como el contenido de un carro de compras, asegurando los datos en caso de accidente o mal uso (cuando la aplicación es abierta en una segunda ventana, por ejemplo).

Por el otro lado, el atributo localStorage nos permite grabar contenidos extensos de información en el ordenador del usuario. La información almacenada es persistente y no expira, excepto por razones de seguridad.

Ambos atributos, sessionStorage y localStorage reemplazan la anterior función del sistema de cookies y fueron creados

para superar sus limitaciones.

La segunda API, agrupada dentro de las APIs de almacenamiento pero independiente del resto, es **Indexed Database**. La función elemental de un sistema de base de datos es la de almacenar información indexada. Web Storage API trabaja sobre el almacenamiento de grandes o pequeñas cantidades de información, datos temporales o permanentes, pero no datos estructurados. Esta es una posibilidad solo disponible para sistemas de base de datos y la razón de la existencia de esta API.

Indexed Database es una sustitución de la API Web SQL Database. Debido a desacuerdos acerca del estándar apropiado a utilizar, ninguna de estas dos APIs ha sido completamente adoptada. De hecho, en este mismo momento, el desarrollo de Web SQL Database API (la cual había sido recibida con brazos abiertos al comienzo), ha sido cancelado.

Debido a que la API Indexed Database, también conocida como **IndexedDB**, luce más prometedora y tiene el apoyo de Microsoft, los desarrolladores de Firefox y Google, será nuestra opción para este libro. Sin embargo, tenga siempre presente que en este momento nuevas implementaciones de SQL están siendo consideradas y la situación podría cambiar en el futuro cercano.

File

Bajo el título de **File**, HTML5 ofrece varias APIs destinadas a operar con archivos. En este momento existen tres disponibles: File, File: Directories & System, y File: Writer.

Gracias a este grupo de APIs, ahora podemos crear y procesar archivos en el ordenador del usuario.

Communication

Algunas API tienen un denominador común que nos permite agruparlas juntas. Este es el caso para **XMLHttpRequest Level 2**, **Cross Document Messaging**, y **Web Sockets**.

Internet ha estado siempre relacionado con comunicaciones, por supuesto, pero algunos asuntos no resueltos hacían el proceso complicado y en ocasiones imposible. Tres problemas específicos fueron abordados en HTML5: la API utilizada para la creación de aplicaciones Ajax no estaba completa y era complicada de implementar a través de distintos navegadores, la comunicación entre aplicaciones no relacionadas era no existía, y no había forma de establecer una comunicación bidireccional efectiva para acceder a información en el servidor en tiempo real.

El primer problema fue resuelto con el desarrollo de **XMLHttpRequest Level 2**. XMLHttpRequest fue la API usada por mucho tiempo para crear aplicaciones Ajax, códigos que acceden al servidor sin recargar la página web. El nivel 2 de esta API incorpora nuevos eventos, provee más funcionalidad (con eventos que permiten hacer un seguimiento del proceso), portabilidad (la API es ahora estándar), y accesibilidad (usando solicitudes cruzadas, desde un dominio a otro).

La solución para el segundo problema fue la creación de **Cross Document Messaging**. Esta API ayuda a los desarrolladores a superar las limitaciones existentes para comunicar diferentes cuadros y ventanas entre sí. Ahora una comunicación segura a través de diferentes aplicaciones es ofrecida por esta API utilizando mensajes.

La solución para el último de los problemas listados anteriormente es **Web Sockets**. Su propósito es proveer las herramientas necesarias para la creación de aplicaciones de red que trabajan en tiempo real (por ejemplo, salas de chat). La API les permite a las aplicaciones obtener y enviar información al servidor en períodos cortos de tiempo, volviendo posible las aplicaciones en tiempo real para la web.

Web Workers

Esta es una API única que expande Javascript a un nuevo nivel. Este lenguaje no es un lenguaje multitarea, lo que significa que solo puede hacerse cargo de una sola tarea a la vez. **Web Workers** provee la posibilidad de procesar código detrás de escena (ejecutado aparte del resto), sin interferir con la actividad en la página web y del código principal. Gracias a esta API Javascript ahora puede ejecutar múltiples tareas al mismo tiempo.

History

Ajax cambió la forma en la que los usuarios interactúan con sitios y aplicaciones web. Y los navegadores no estaban preparados para esta situación. **History** fue implementada para adaptar las aplicaciones modernas a la forma en que los navegadores hacen seguimiento de la actividad del usuario. Esta API incorpora técnicas para generar artificialmente URLs por cada paso en el proceso, ofreciendo la posibilidad de retorna a estados previos de la aplicación utilizando procedimientos

estándar de navegación.

Offline

Incluso hoy día, con acceso a Internet en cada lugar que vamos, quedar desconectado es aún posible. Dispositivos portátiles se encuentran en todas partes, pero no la señal para establecer comunicación. Y los ordenadores de escritorio también pueden dejarnos desconectados en los momentos más críticos. Con la combinación de atributos HTML, eventos controlados por Javascript y archivos de texto, **Offline** permitirá a las aplicaciones trabajar en línea o desconectados, de acuerdo a la situación del usuario.

4.6 Librerías externas

HTML5 fue desarrollado para expandir la web utilizando un set de tecnologías estándar que todo navegador pueda entender y procesar. Y fue creado para proveer todas las herramientas que un desarrollador necesita. De hecho, HTML5 fue conceptualizado para no depender de tecnologías de terceras partes. Pero por una razón u otra siempre necesitaremos contar con ayuda extra.

Antes de la aparición de HTML5, varias librerías Javascript fueron desarrolladas para superar las limitaciones de las tecnologías disponibles al momento. Algunas de estas librerías tenían propósitos específicos (desde procesamiento y validación de formularios hasta generación y manipulación de gráficos). Algunas se volvieron extremadamente populares y otras son casi imposibles de imitar por desarrolladores independientes (como es el caso de Google Maps).

Incluso cuando futuras implementaciones provean mejores métodos o funciones, los programadores siempre encontrarán una manera más fácil de lidiar con un asunto determinado. Librerías desarrolladas por terceros que facilitan tareas complicadas siempre estarán vivas y creciendo en número.

Estas librerías no son parte de HTML5 pero son una parte importante de la web y algunas de ellas son actualmente usadas en sitios web y aplicaciones exitosas. Junto con el resto de las funciones incorporadas por esta especificación, mejoran Javascript y acercan las tecnologías de última generación al público en general.

jQuery

Esta es la librería web más popular disponible en estos días. La librería jQuery es gratuita y fue diseñada para simplificar la creación de sitios web modernos. Facilita la selección de elementos HTML, la creación de animaciones y efectos, y también controla eventos y ayuda a implementar Ajax en nuestras aplicaciones.

La librería jQuery se encuentra en un archivo pequeño que se puede descargar desde **www.jquery.com** y luego incluir en nuestros documentos usando la etiqueta `<script>`. Provee una API sencilla que cualquiera puede aprender y rápidamente aplicar a sus proyectos.

Una vez que el archivo provisto por jQuery es incluido en nuestro documento, ya estamos listos para aprovechar los métodos simples incorporados por la librería y convertir nuestra web estática en una moderna y práctica aplicación.

jQuery tiene la ventaja de proveer soporte para viejos navegadores y vuelve simple tareas cotidianas. Puede ser utilizado junto con HTML5 o como una forma simple de reemplazar funciones de HTML5 en navegadores que no están preparados para esta tecnología.

Google Maps

Accesible por medio de Javascript (y otras tecnologías), Google Maps es un complejo y único set de herramientas que nos permite desarrollar cualquier servicio de mapeado para la web que podamos imaginar. Google se ha vuelto el líder en esta clase de servicios y a través de la tecnología Google Maps provee acceso a un extremadamente preciso y detallado mapa del mundo. Utilizando esta API podemos encontrar lugares específicos, calcular distancias, hallar sitios populares o incluso obtener una vista del lugar seleccionado como si estuviéramos presentes.

Google Maps es gratuita y disponible para todo desarrollador. Diferentes versiones de la API se pueden encontrar en: *code.google.com/apis/maps/.*

4.7 Referencia rápida

En HTML5, Javascript fue mejorado por medio de la adición de nuevas funciones y la incorporación de métodos nativos.

Elementos

<script> Este elemento ahora tiene a Javascript como el lenguaje por defecto. El atributo **`type`** ya no es necesario.

Selectores

La posibilidad de seleccionar un elemento del documento dinámicamente desde código Javascript se ha vuelto esencial para cualquier aplicación web. Nuevos métodos han sido incorporados con este propósito.

getElementsByClassName Este selector nos permite encontrar elementos en el documento por medio del valor de su atributo **`class`**. Es una adición a los ya conocidos **`getElementsByTagName`** y **`getElementById`**.

querySelector(selectores) Este método usa selectores CSS para referenciar elementos en el documento. Los selectores son declarados entre paréntesis. Este método puede ser combinado con otros para construir referencias más específicas. Retorna solo el primer elemento encontrado.

querySelectorAll(selectores) Este método es similar a **`querySelector()`** pero retorna todos los elementos que concuerdan con los selectores especificados.

Eventos

La relevancia de los eventos en las aplicaciones web motivó la estandarización de métodos ya disponibles en navegadores líderes.

addEventListener(evento, manejador, captura) Este método es usado para agregar una escucha para un evento. El método recibe tres valores: el nombre del evento, la función que responderá al evento, y un valor booleano (verdadero o falso) que indica el orden de ejecución de varios eventos disparados al mismo tiempo. Normalmente el tercer atributo es configurado como **`false`**.

removeEventListener(evento, manejador, captura) Este método es usado para remover una escucha para un evento, desactivando el manejador. Los valores necesarios son los mismos que los usados para **`addEventListener()`**.

APIs

El alcance de Javascript ha sido expandido con un grupo de poderosas librerías accesibles a través de interfaces llamadas APIs.

Canvas Esta API es una API de dibujo, específica para la creación y manipulación de gráficos. Utiliza métodos Javascript predefinidos para operar.

Drag and Drop Esta API hace que arrastrar y soltar elementos con el ratón en la pantalla sea posible también en la web.

Geolocation Esta API tiene la intención de proveer acceso a información correspondiente con la ubicación física del dispositivo que está accediendo a la aplicación. Puede retornar datos como la latitud y longitud utilizando diferentes mecanismos (como información de la red o GPS).

Web Storage Esta API introduce dos atributos para almacenar datos en el ordenador del usuario: **`sessionStorage`** y **`localStorage`**. El atributo **`sessionStorage`** permite a los desarrolladores hacer un seguimiento de la actividad de los usuarios almacenando información que estará disponible en cada instancia de la aplicación durante la duración de la sesión. El atributo **`localStorage`**, por otro lado, ofrece a los desarrolladores un área de almacenamiento, creada para cada aplicación, que puede conservar varios megabytes de información, preservando de este modo información y datos en el ordenador del usuario de forma persistente.

Indexed Database Esta API agrega la capacidad de trabajar con bases de datos del lado del usuario. El sistema fue desarrollado independientemente de previas tecnologías y provee una base de datos destinada a aplicaciones web. La base de datos es almacenada en el ordenador del usuario, los datos son persistentes y, por supuesto, son exclusivos de la aplicación que los creó.

File Este es un grupo de APIs desarrollada para proveer la capacidad de leer, escribir y procesar archivos de usuario.

XMLHttpRequest Level 2 Esta API es una mejora de la vieja XMLHttpRequest destinada a la construcción de aplicaciones Ajax. Incluye nuevos métodos para controlar el progreso de la operación y realizar solicitudes cruzadas (desde diferentes orígenes).

Cross Document Messaging Esta API introduce una nueva tecnología de comunicación que permite a aplicaciones comunicarse entre sí a través de diferentes cuadros o ventanas.

WebSockets Esta API provee un mecanismo de comunicación de dos vías entre clientes y servidores para generar aplicaciones en tiempo real como salas de chat o juegos en línea.

Web Workers Esta API potencia Javascript permitiendo el procesamiento de código detrás de escena, de forma separada del código principal, sin interrumpir la actividad normal de la página web, incorporando la capacidad de multitarea a este lenguaje.

History Esta API provee la alternativa de incorporar cada paso en el proceso de una aplicación dentro del historial de navegación del navegador.

Offline Esta API apunta a mantener las aplicaciones funcionales incluso cuando el dispositivo es desconectado de la red.

Capítulo 5
Video y audio

5.1 Reproduciendo video con HTML5

Una de las características más mencionadas de HTML5 fue la capacidad de procesar video. El entusiasmo nada tenía que ver con las nuevas herramientas provistas por HTML5 para este propósito, sino más bien con el hecho de que desde los videos se volvieron una pieza esencial de Internet, todos esperaban soporte nativo por parte de los navegadores. Era como que todos conocían la importancia de los videos excepto aquellos encargados de desarrollar las tecnologías para la web.

Pero ahora que ya disponemos de soporte nativo para videos e incluso un estándar que nos permitirá crear aplicaciones de procesamiento de video compatibles con múltiples navegadores, podemos comprender que la situación era mucho más complicada de lo que nos habíamos imaginado. Desde codificadores hasta consumo de recursos, las razones para no implementar video de forma nativa en los navegadores eran mucho más complejas que los códigos necesarios para hacerlo.

A pesar de estas complicaciones, HTML5 finalmente introdujo un elemento para insertar y reproducir video en un documento HTML. El elemento **<video>** usa etiquetas de apertura y cierre y solo unos pocos parámetros para lograr su función. La sintaxis es extremadamente sencilla y solo el atributo **src** es obligatorio:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Video</title>
</head>
<body>
<section id="reproductor">
  <video src="http://minkbooks.com/content/trailer.mp4" controls>
  </video>
</section>
</body>
</html>
```

Listado 5-1. *Sintaxis básica para el elemento* <video>.

En teoría, el código del Listado 5-1 debería ser más que suficiente. Repito, en teoría. Pero como explicamos anteriormente, las cosas se vuelven un poco más complicadas en la vida real. Primero debemos proveer al menos dos archivos diferentes con formatos de video diferentes: OGG y MP4. Esto es debido a que a pesar de que el elemento **<video>** y sus atributos son estándar, no existe un formato estándar de video. Primero, algunos navegadores soportan un codificador de video que otros no, y segundo el codificador utilizado en el formato MP4 (el único soportado por importantes navegadores como Safari e Internet Explorer) se encuentra bajo licencia comercial.

Los formatos OGG y MP4 son contenedores de video y audio. OGG contiene codificadores de video Theora y de audio Vorbis, y los disponibles para el contenedor MP4 son H.264 para video y AAC para audio. En este momento OGG es reconocido por Firefox, Google Chrome y Opera, mientras que MP4 trabaja en Safari, Internet Explorer y también Google Chrome.

El elemento <video>

Intentemos ignorar por un momento estas complicaciones y disfrutar de la simplicidad del elemento **<video>**. Este elemento ofrece varios atributos para establecer su comportamiento y configuración. Los atributos **width** y **height**, al igual que en otros elementos HTML ya conocidos, declaran las dimensiones para el elemento o ventana del reproductor. El tamaño del video será automáticamente ajustado para entrar dentro de estos valores, pero no fueron considerados para redimensionar el video sino limitar el área ocupada por el mismo para mantener consistencia en el diseño. El atributo **src** especifica la fuente del video. Este atributo puede ser reemplazado por el elemento **<source>** y su propio atributo **src** para declarar varias fuentes con diferentes formatos, como en el siguiente ejemplo:

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
  <title>Reproductor de Video</title>
</head>
<body>
<section id="reproductor">
  <video id="medio" width="720" height="400" controls>
    <source src="http://minkbooks.com/content/trailer.mp4">
    <source src="http://minkbooks.com/content/trailer.ogg">
  </video>
</section>
</body>
</html>
```

Listado 5-2. *Reproductor de video con controles por defecto y compatible con navegadores HTML5.*

En el Listado 5-2, el elemento **<video>** fue expandido. Ahora, dentro de las etiquetas del elemento hay dos elementos **<source>**. Estos nuevos elementos proveen diferentes fuentes de video para que los navegadores puedan elegir. El navegador leerá la etiqueta **<source>** y decidirá cual archivo reproducir de acuerdo a los formatos soportados (MP4 u OGG).

Hágalo usted mismo: Cree un nuevo archivo HTML vacío con el nombre **video.html** (o cualquier otro nombre que desee), copie el código del Listado 5-2, y abra el archivo en diferentes navegadores para comprobar el modo en que el elemento **<video>** trabaja en cada uno de ellos.

Atributos para <video>

Incluimos un atributo en la etiqueta **<video>** en los Listados 5-1 y 5-2 que probablemente llamó su atención. El atributo **controls** es uno de varios atributos disponibles para este elemento. Éste, en particular, muestra controles de video provistos por el navegador por defecto. Cuando el atributo está presente cada navegador activará su propia interface, permitiendo al usuario comenzar a reproducir el video, pausarlo o saltar hacia un cuadro específico, entre otras funciones.

Junto con **controls**, también podemos usar los siguientes:

autoplay Cuando este atributo está presente, el navegador comenzará a reproducir el video automáticamente tan pronto como pueda.

loop Si este atributo es especificado, el navegador comenzará a reproducir el video nuevamente cuando llega al final.

poster Este atributo es utilizado para proveer una imagen que será mostrada mientras esperamos que el video comience a ser reproducido.

preload Este atributo puede recibir tres valores distintos: **none**, **metadata** o **auto**. El primero indica que el video no debería ser cacheado, por lo general con el propósito de minimizar tráfico innecesario. El segundo valor, **metadata**, recomendará al navegador que trate de capturar información acerca de la fuente (por ejemplo, dimensiones, duración, primer cuadro, etc…). El tercer valor, **auto**, es el valor configurado por defecto que le sugerirá al navegador descargar el archivo tan pronto como sea posible.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Video</title>
</head>
<body>
  <section id="reproductor">
    <video id="medio" width="720" height="400" preload controls
          loop poster="http://minkbooks.com/content/poster.jpg">
      <source src="http://minkbooks.com/content/trailer.mp4">
      <source src="http://minkbooks.com/content/trailer.ogg">
    </video>
  </section>
</body>
</html>
```

Listado 5-3: *Aprovechando los atributos del elemento* <video>.

En el Listado 5-3, el elemento **<video>** fue poblado con atributos. Debido a las diferencias en comportamiento entre un

navegador y otro, algunos atributos estarán habilitados o deshabilitados por defecto, y algunos de ellos incluso no trabajarán en algunos navegadores o bajo determinadas circunstancias. Para obtener un control absoluto sobre el elemento **<video>** y el medio reproducido, deberemos programar nuestro propio reproductor de video en Javascript aprovechando los nuevos métodos, propiedades y eventos incorporados en HTML5.

5.2 Programando un reproductor de video

Si ha probado los anteriores códigos en diferentes navegadores, seguramente habrá notado que los diseños gráficos de los controles del reproductor difieren de uno a otro. Cada navegador tiene sus propios botones y barras de progreso, e incluso sus propias funciones. Esta situación puede ser aceptable en algunas circunstancias pero en un ambiente profesional, donde cada detalle cuenta, resulta absolutamente necesario que un diseño consistente sea preservado a través de dispositivos y aplicaciones, y también disponer de un control absoluto sobre todo el proceso.

HTML5 proporciona nuevos eventos, propiedades y métodos para manipular video e integrarlo al documento. De ahora en más, podremos crear nuestro propio reproductor de video y ofrecer las funciones que queremos usando HTML, CSS y Javascript. El video es ahora parte integral del documento.

El diseño

Todo reproductor de video necesita un panel de control con al menos algunas funciones básicas. En la nueva plantilla del Listado 5-4, un elemento **<nav>** fue agregado luego de **<video>**. Este elemento **<nav>** contiene dos elementos **<div>** (**botones** y **barra**) para ofrecer un botón "Reproducir" y una barra de progreso.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Video</title>
  <link rel="stylesheet" href="reproductor.css">
  <script src="reproductor.js"></script>
</head>
<body>
<section id="reproductor">
  <video id="medio" width="720" height="400">
    <source src="http://minkbooks.com/content/trailer.mp4">
    <source src="http://minkbooks.com/content/trailer.ogg">
  </video>
  <nav>
    <div id="botones">
      <button type="button" id="reproducir">Reproducir</button>
    </div>
    <div id="barra">
      <div id="progreso"></div>
    </div>
    <div style="clear: both"></div>
  </nav>
</section>
</body>
</html>
```

Listado 5-4. *Plantilla HTML para nuestro reproductor de video.*

Además del video, esta plantilla también incluye dos archivos para acceder a códigos externos. Uno de ellos es **player.css** para los siguientes estilos CSS:

```
body{
  text-align: center;
}
header, section, footer, aside, nav, article, figure, figcaption,
                                                           hgroup{
  display : block;
}
#reproductor{
  width: 720px;
  margin: 20px auto;
  padding: 5px;
```

```
  background: #999999;
  border: 1px solid #666666;

  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
nav{
  margin: 5px 0px;
}

#botones{
  float: left;
  width: 100px;
  height: 20px;
}
#barra{
  position: relative;
  float: left;
  width: 600px;
  height: 16px;
  padding: 2px;
  border: 1px solid #CCCCCC;
  background: #EEEEEE;
}
#progreso{
  position: absolute;
  width: 0px;
  height: 16px;
  background: rgba(0,0,150,.2);
}
```

Listado 5-5. *Estilos CSS para el reproductor.*

El código del Listado 5-5 usa técnicas del Modelo de Caja Tradicional estudiado en el Capítulo 2 para crear la caja que contiene cada pieza del reproductor de video y ubicarla en el centro de la ventana. No hay nuevas propiedades o sorpresas en este código, es solo un grupo de propiedades CSS ya estudiadas y conocidas para proveer estilos a los elementos del reproductor. Sin embargo, existen dos propiedades que pueden resultar inusuales. La propiedad **`position`**, conocida por viejos programadores CSS, fue usada para superponer un elemento sobre otro (**`barra`** y **`progreso`**). Y la propiedad **`width`**, para el elemento **`<div>`** identificado como **`progreso`**, fue inicializada en **`0`**. Esto se debe a que el elemento será utilizado para simular una barra de progreso que cambiará de tamaño a medida que el video es reproducido, y que, por supuesto, comenzará a crecer desde **`0`**.

> **Hágalo usted mismo:** Copie la nueva plantilla del Listado 5-4 en el archivo HTML (**`video.html`**). Cree dos nuevos archivos vacíos para los estilos CSS y el código Javascript. Estos archivos deberían ser llamados **`reproductor.css`** y **`reproductor.js`** respectivamente. Copie el código del Listado 5-5 dentro del archivo correspondiente y luego haga lo mismo para cada código Javascript listado de ahora en adelante.

El código

Es momento de escribir el código Javascript para nuestro reproductor. Existen diferentes formas de programar un reproductor de video, pero en este capítulo vamos solo a explicar cómo aplicar los necesarios eventos, métodos y propiedades para procesamiento básico de video. El resto quedará librado a su imaginación.

Para nuestro propósito, vamos a trabajar con unas pocas funciones simples que nos permitirán reproducir y pausar el video, mostrar una barra de progreso mientras el video es reproducido y ofrecer la opción de hacer clic sobre esta barra para adelantar o retroceder el video.

Los eventos

HTML5 incorpora nuevos eventos que son específicos de cada API. Para el procesamiento de video y audio, por ejemplo, los eventos fueron incorporados con el objetivo de informar sobre la situación del medio (el progreso de la descarga, si la reproducción del medio finalizó, o si la reproducción del medio es comenzada o pausada, entre otras). No vamos a utilizarlos

en nuestros ejemplos pero serán necesarios para construir aplicaciones complejas. Estos son los más relevantes:

progress Este evento es disparado periódicamente para informar acerca del progreso de la descarga del medio. La información estará disponible a través del atributo **buffered**, como veremos más adelante.

canplaythrough Este evento es disparado cuando el medio completo puede ser reproducido sin interrupción. El estado es establecido considerando la actual tasa de descarga y asumiendo que seguirá siendo la misma durante el resto del proceso. Existe otro evento más para este propósito, **canplay**, pero no considera toda la situación y es disparado tan pronto como algunas partes del medio se encuentran disponibles (luego de descargar los primeros cuadros de un video, por ejemplo).

ended Es disparado cuando el reproductor llega al final del medio.

pause Es disparado cuando el reproductor es pausado.

play Es disparado cuando el medio comienza a ser reproducido.

error Este evento es disparado cuando ocurre un error. Es relacionado con el elemento **<source>** correspondiente a la fuente del medio que produjo el error.

Para nuestro reproductor de ejemplo solo vamos a escuchar a los habituales eventos **click** y **load**.

IMPORTANTE: Eventos, métodos y propiedades para APIs están aún en proceso de desarrollo. En este libro vamos a estudiar solo aquellos que consideramos relevantes e indispensables para nuestros ejemplos. Para ver cómo la especificación está progresando con respecto a esto, visite nuestro sitio web y siga los enlaces correspondientes a cada capítulo.

```
function iniciar() {
  maximo=600;
  medio=document.getElementById('medio');
  reproducir=document.getElementById('reproducir');
  barra=document.getElementById('barra');
  progreso=document.getElementById('progreso');

  reproducir.addEventListener('click', presionar, false);
  barra.addEventListener('click', mover, false);
}
```

Listado 5-6. *Función inicial.*

El Listado 5-6 presenta la primera función de nuestro reproductor de video. La función fue llamada **iniciar** debido a que será la función que iniciará la ejecución de la aplicación tan pronto como el documento sea completamente cargado.

Debido a que esta es la primera función a ser ejecutada, necesitamos definir unas variables globales para configurar nuestro reproductor. Usando el selector **getElementById** creamos una referencia a cada uno de los elementos del reproductor para poder acceder a ellos en el resto del código más adelante. También declaramos la variable **maximo** para conocer siempre el máximo tamaño posible para la barra de progreso (600 pixeles).

Hay dos acciones a las que tenemos que prestar atención desde el código: cuando el usuario hace clic sobre el botón “Reproducir” y cuando hace clic sobre la barra de progreso para avanzar o retroceder el video. Dos escuchas para el evento **click** fueron agregadas con el propósito de controlar estas situaciones. Primero agregamos la escucha al elemento **reproducir** que ejecutará la función **presionar()** cada vez que el usuario haga clic sobre el botón “Reproducir”. La otra escucha es para el elemento **barra**. En este caso, la función **mover()** será ejecutada cada vez que el usuario haga clic sobre la barra de progreso.

Los métodos

La función **presionar()** incorporada en el Listado 5-7 es la primera función que realmente realiza una tarea. Esta función ejecutará de acuerdo a la situación actual dos métodos específicos de esta API: **play()** y **pause()**:

```
function presionar(){
  if(!medio.paused && !medio.ended) {
    medio.pause();
    reproducir.innerHTML='Reproducir';
    window.clearInterval(bucle);
```

```
    }else{
      medio.play();
      reproducir.innerHTML='Pausa';
      bucle=setInterval(estado, 1000);
    }
  }
```

Listado 5-7. *Esta función inicia y pausa la reproducción del video.*

Los métodos **`play()`** y **`pause()`** son parte de una lista de métodos incorporados por HTML5 para procesamiento de medios. Los siguientes son los más relevantes:

play() Este método comienza a reproducir el medio desde el inicio, a menos que el medio haya sido pausado previamente.

pause() Este método pausa la reproducción.

load() Este método carga el archivo del medio. Es útil en aplicaciones dinámicas para cargar el medio anticipadamente.

canPlayType(formato) Con este método podemos saber si el formato del archivo es soportado por el navegador o no.

Las propiedades

La función **`presionar()`** también usa unas pocas propiedades para recabar información sobre el medio. Las siguientes son las más relevantes:

paused Esta propiedad retorna **`true`** (verdadero) si la reproducción del medio está actualmente pausada o no a comenzado.

ended Esta propiedad retorna **`true`** (verdadero) si la reproducción del medio ha finalizado porque se llegó al final.

duration Esta propiedad retorna la duración del medio en segundos.

currentTime Esta es una propiedad que puede retornar o recibir un valor para informar sobre la posición en la cual el medio está siendo reproducido o especifica una nueva posición donde continuar reproduciendo.

error Esta propiedad retorna el valor del error ocurrido.

buffered Esta propiedad ofrece información sobre la parte del archivo que ya fue cargada en el buffer. Nos permite crear un indicador para mostrar el progreso de la descarga. La propiedad es usualmente leída cuando el evento **`progress`** es disparado. Debido a que los usuarios pueden forzar al navegador a cargar el medio desde diferentes posiciones en la línea de tiempo, la información retornada por **`buffered`** es un array conteniendo cada parte del medio que ya fue descargada, no solo la que comienza desde el principio. Los elementos del array son accesibles por medio de los atributos **`end()`** y **`start()`**. Por ejemplo, el código **`buffered.end(0)`** retornará la duración en segundos de la primera porción del medio encontrada en el buffer. Esta propiedad y sus atributos están bajo desarrollo en este momento.

El código en operación

Ahora que ya conocemos todos los elementos involucrados en el procesamiento de video, echemos un vistazo a cómo trabaja la función **`presionar()`**.

Esta función es ejecutada cuando el usuario presiona el botón “Reproducir” en nuestro reproductor. Este botón tendrá dos propósitos: mostrará el mensaje “Reproducir” para reproducir el video o “Pausa” para detenerlo, de acuerdo a las circunstancias. Por lo tanto, cuando el video fue pausado o no comenzó, presionar este botón comenzará o continuará la reproducción. Lo opuesto ocurrirá si el video está siendo reproducido, entonces presionar el botón pausará el video.

Para lograr esto el código detecta la situación del medio comprobando el valor de las propiedades **`paused`** y **`ended`**. En la primera línea de la función tenemos un condicional **`if`** para este propósito. Si el valor de **`medio.paused`** y **`medio.ended`** es falso, significará que el video está siendo reproducido, entonces el método **`pause()`** es ejecutado para pausar el video y el texto del botón es cambiado a “Reproducir” usando **`innerHTML`**.

Si lo opuesto ocurre, el video fue pausado previamente o terminó de ser reproducido, entonces la condición será falsa (**`medio.paused`** o **`medio.ended`** es verdadero) y el método **`play()`** es ejecutado para comenzar o restaurar la reproducción del video. En este caso también realizamos una importante acción que es configurar un intervalo usando **`setInterval()`** para ejecutar la función **`estado()`** una vez por segundo de ahora en más.

```
function estado(){
  if(!medio.ended){
    var total=parseInt(medio.currentTime*maximo/medio.duration);
    progreso.style.width=total+'px';
  }else{
    progreso.style.width='0px';
    reproducir.innerHTML='Reproducir';
    window.clearInterval(bucle);
  }
}
```

Listado 5-8. *Esta función actualiza la barra de progreso una vez por segundo.*

La función **estado()** en el Listado 5-8 es ejecutada cada segundo mientras el video es reproducido. También utilizamos un condicional **if** en esta función para controlar el estado del video. Si la propiedad **ended** retorna falso, calculamos qué tan larga la barra de progreso debe ser en pixeles y asignamos el valor al elemento **<div>** que la representa. En caso de que la propiedad sea verdadera (lo cual significa que la reproducción del video ha terminado), retornamos el valor de la barra de progreso a 0 pixeles, cambiamos el botón a "Reproducir", y cancelamos el intervalo usando **clearInterval**. En este caso la función **estado()** no será ejecutada nunca más.

Volvamos unos pasos para estudiar cómo calculamos el tamaño de la barra de progreso. Debido a que la función **estado()** será ejecutada cada segundo mientras el video se está reproduciendo, el valor del tiempo en el que el video se encuentra cambiará constantemente. Este valor en segundos es obtenido de la propiedad **currentTime**. También contamos con el valor de la duración del video en la propiedad **duration**, y el máximo tamaño de la barra de progreso en la variable **maximo** que definimos al principio. Con estos tres valores podemos calcular cuántos pixeles de largo la barra debería ser para representar los segundos ya reproducidos. La fórmula **tiempo actual × maximo / duración total** transformará los segundos en pixeles para cambiar el tamaño del elemento **<div>** que representa la barra de progreso.

La función para responder al evento **click** del elemento **reproducir** (el botón) ya fue creada. Ahora es tiempo de hacer lo mismo para responder a los clics hechos sobre la barra de progreso:

```
function mover(e){
  if(!medio.paused && !medio.ended){
    var ratonX=e.pageX-barra.offsetLeft;
    var nuevoTiempo=ratonX*medio.duration/maximo;
    medio.currentTime=nuevoTiempo;
    progreso.style.width=ratonX+'px';
  }
}
```

Listado 5-9. *Comenzar a reproducir desde la posición seleccionada por el usuario.*

Una escucha para el evento **click** fue agregada al elemento **barra** para responder cada vez que el usuario quiera comenzar a reproducir el video desde una nueva posición. La escucha usa la función **mover()** para responder al evento cuando es disparado. Puede ver esta función en el Listado 5-9. Comienza con un **if**, al igual que las anteriores funciones, pero esta vez el objetivo es controlar que la acción se realice sólo cuando el video está siendo reproducido. Si las propiedades **paused** y **ended** son falsas significa que el video está siendo reproducido y el código tiene que ser ejecutado.

Debemos hacer varias cosas para calcular el tiempo en el cual el video debería comenzar a ser reproducido. Necesitamos determinar cuál era la posición del ratón cuando el clic sobre la barra fue realizado, cuál es la distancia en pixeles desde esa posición hasta el comienzo de la barra de progreso y cuantos segundos esa distancia representa en la línea de tiempo.

Los procesos para agregar una escucha (o registrar un evento), tales como **addEventListener()**, siempre envían un valor que hacer referencia al evento. Esta referencia es enviada como un atributo a la función que responde al evento. Tradicionalmente la variable **e** es usada para almacenar este valor. En la función del Listado 5-9 usamos esta variable y la propiedad **pageX** para capturar la posición exacta del puntero del ratón al momento en el que el clic fue realizado. El valor retornado por **pageX** es relativo a la página, no a la barra de progreso o la ventana. Para saber cuántos pixeles hay desde el comienzo de la barra de progreso y la posición del puntero, tenemos que substraer el espacio entre el lado izquierdo de la página y el comienzo de la barra. Recuerde que la barra está localizada en una caja que se encuentra centrada en la ventana. Los valores dependerán de cada situación en particular, por lo tanto supongamos que la barra está localizada a 421 pixeles del lado izquierdo de la página web y el clic fue realizado en el medio de la barra. Debido a que la barra tiene una longitud de 600 pixeles, el clic fue hecho a 300 pixeles desde el comienzo de la barra. Sin embargo, la propiedad **pageX** no retornará el

valor 300, sino 721. Para obtener la posición exacta en la barra donde el clic ocurrió, debemos substraer de **pageX** la distancia desde el lado izquierdo de la página hasta el comienzo de la barra (en nuestro ejemplo, 421 pixeles). Esta distancia puede ser obtenida mediante la propiedad **offsetLeft**. Entonces, usando la formula **e.pageX - barra.offsetLeft** conseguimos exactamente la posición del puntero del ratón relativa al comienzo de la barra. En nuestro ejemplo, la fórmula en números sería: **721 – 421 = 300**.

Una vez obtenido este valor, debemos convertirlo a segundos. Usando la propiedad **duration**, la posición exacta del puntero del ratón en la barra y el tamaño máximo de la barra construimos la fórmula **ratonX × video.duration / maximo** y almacenamos el resultado dentro de la variable **nuevoTiempo**. Este resultado es el tiempo en segundos que la posición del puntero del ratón representa en la línea de tiempo.

El siguiente paso es comenzar a reproducir el video desde la nueva posición. La propiedad **currentTime**, como ya mencionamos, retorna la posición actual del video en segundos pero también avanza o retrocede el video a un tiempo específico si un nuevo valor le es asignado. Con el código **medio.currentTime=nuevoTiempo** movemos el video a la posición deseada.

Lo único que resta por hacer es cambiar el tamaño del elemento **progreso** para reflejar en pantalla la nueva situación. Utilizando el valor de la variable **ratonX** cambiamos el tamaño del elemento para alcanzar exactamente la posición done el clic fue hecho.

El código para nuestro reproductor de video ya está casi listo. Tenemos todos los eventos, métodos, propiedades y funciones que nuestra aplicación necesita. Solo hay una cosa más que debemos hacer, un evento más que debemos escuchar para poner nuestro código en marcha:

```
window.addEventListener('load', iniciar, false);
```

Listado 5-10. *Escuchando al evento* `load`.

Podríamos haber usado la técnica **window.onload** para registrar el manejador del evento, y de hecho hubiese sido la mejor opción para hacer nuestros códigos compatibles con viejos navegadores. Sin embargo, debido a que este libro es acerca de HTML5, decidimos usar el nuevo estándar **addEventListener()**.

Hágalo usted mismo: Copie todos los códigos Javascript desde el Listado 5-6 dentro del archivo **reproductor.js**. Abra el archivo **video.html** con la plantilla del Listado 5-4 en su navegador y haga clic en el botón “Reproducir”. Intente utilizar la aplicación desde diferentes navegadores.

5.3 Formatos de video

Por el momento no existe un estándar para formatos de video y audio en la web. Existen varios contenedores y diferentes codificadores disponibles, pero ninguno fue totalmente adoptado y no hay consenso alguno de parte de los fabricantes de navegadores para lograr un estándar en el futuro cercano.

Los contenedores más comunes son OGG, MP4, FLV y el nuevo propuesto por Google, WEBM. Normalmente estos contenedores contienen video codificado con los codificadores Theora, H.264, VP6 o VP8, respectivamente. Esta es la lista de los más usados:

- **OGG** codificador de video Theora y audio Vorbis.
- **MP4** codificador de video H.264 y audio AAC.
- **FLV** codificador de video VP6 y audio MP3. También soporta H.264 y AAC.
- **WEBM** codificador de video VP8 y audio Vorbis.

Los codificadores utilizados para OGG y WEBM son gratuitos, pero los utilizados para MP4 y FLV están patentados, lo que significa que si queremos usar MP4 o FLV para nuestras aplicaciones deberemos pagar. Algunas restricciones son anuladas para aplicaciones gratuitas.

El tema es que en este momento Safari e Internet Explorer no soportan la tecnología gratuita. Ambos solo trabajan con MP4 y solo Internet Explorer anunció la inclusión del codificador VP8 en el futuro. Esta es la lista de los navegadores más populares:

- **Firefox** codificador de video Theora y audio Vorbis.
- **Google Chrome** codificador de video Theora y audio Vorbis. También soporta codificador de video H.264 y audio AAC.
- **Opera** codificador de video Theora y audio Vorbis.
- **Safari** codificador de video H.264 y audio AAC.
- **Internet Explorer** codificador de video H.264 y audio AAC.

Un mayor soporte para el formato WEBM en el futuro podría mejorar la situación, pero probablemente no habrá un formato estándar por al menos los próximos dos o tres años y tendremos que considerar diferentes alternativas de acuerdo a la naturaleza de nuestra aplicación y nuestro negocio.

5.4 Reproduciendo audio con HTML5

Audio no es un medio tan popular como video en Internet. Podemos filmar un video con una cámara personal que generará millones de vistas en sitios web como www.youtube.com, pero crear un archivo de audio que obtenga el mismo resultado es prácticamente imposible. Sin Embargo, el audio se encuentra aún disponible, ganando su propio mercado en shows de radio y podcasts en toda la red.

HTML5 provee un nuevo elemento para reproducir audio en un documento HTML. El elemento, por supuesto, es **<audio>** y comparte casi las mismas características del elemento **<video>**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Audio</title>
</head>
<body>
<section id="reproductor">
  <audio src=”http://minkbooks.com/content/beach.mp3” controls>
  </audio>
</section>
</body>
</html>
```

Listado 5-11. *HTML básico para reproducir audio.*

El elemento <audio>

El elemento **<audio>** trabaja del mismo modo y comparte varios atributos con el elemento **<video>**:

src Este atributo especifica la URL del archivo a ser reproducido. Al igual que en el elemento **<video>** normalmente será reemplazado por el elemento **<source>** para ofrecer diferentes formatos de audio entre los que el navegador pueda elegir.

controls Este atributo activa la interface que cada navegador provee por defecto para controlar la reproducción del audio.

autoplay Cuando este atributo está presente, el audio comenzará a reproducirse automáticamente tan pronto como sea posible.

loop Si este atributo es especificado, el navegador reproducirá el audio una y otra vez de forma automática.

preload Este atributo puede tomar tres valores diferentes: **none**, **metadata** o **auto**. El primero indica que el audio no debería ser cacheado, normalmente con el propósito de minimizar tráfico innecesario. El segundo valor, **metadata**, recomendará al navegador obtener información sobre el medio (por ejemplo, la duración). El tercer valor, **auto**, es el valor configurado por defecto y le aconseja al navegador descargar el archivo tan pronto como sea posible.

Una vez más debemos hablar acerca de codificadores, y otra vez debemos decir que el código en el Listado 5-11 debería ser más que suficiente para reproducir audio en nuestro documento, pero no lo es. MP3 está bajo licencia comercial, por lo que no es soportado por navegadores como Firefox u Opera. Vorbis (el codificador de audio del contenedor OGG) es soportado por esos navegadores, pero no por Safari e Internet Explorer. Por esta razón, nuevamente debemos aprovechar el elemento **<source>** para proveer al menos dos formatos entre los cuales el navegador pueda elegir:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Audio</title>
</head>
<body>
<section id="reproductor">
  <audio id="medio" controls>
    <source src="http://minkbooks.com/content/beach.mp3">
    <source src="http://minkbooks.com/content/beach.ogg">
  </audio>
```

```
</section>
</body>
</html>
```

Listado 5-12: *dos fuentes para el mismo audio*

El código en el Listado 5-12 reproducirá música en todos los navegadores utilizando los controles por defecto. Aquellos que no puedan reproducir MP3 reproducirán OGG y viceversa. Recuerde que MP3, al igual que MP4 para video, tienen uso restringido por licencias comerciales, por lo que solo podemos usarlos en circunstancias especiales, de acuerdo con lo determinado por cada licencia.

El soporte para los codificadores de audio libres y gratuitos (como Vorbis) se está expandiendo, pero llevará tiempo transformar este formato desconocido en un estándar.

5.5 Programando un reproductor de audio

La API para medios fue desarrollada tanto para video como para audio. Cada evento, método y propiedad incorporada para video funcionará también con audio. Debido a esto, solo necesitamos reemplazar el elemento **<video>** por el elemento **<audio>** en nuestra plantilla e instantáneamente obtenemos un reproductor de audio:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Reproductor de Audio</title>
  <link rel="stylesheet" href="reproductor.css">
  <script src="reproductor.js"></script>
</head>
<body>
<section id="reproductor">
  <audio id="medio">
    <source src="http://minkbooks.com/content/beach.mp3">
    <source src="http://minkbooks.com/content/beach.ogg">
  </audio>
  <nav>
    <div id="botones">
      <button type="button" id="reproducir">Reproducir</button>
    </div>
    <div id="barra">
      <div id="progreso"></div>
    </div>
    <div style="clear: both"></div>
  </nav>
</section>
</body>
</html>
```

Listado 5-13. *Plantilla para el reproductor de audio.*

En la nueva plantilla del listado 5-13, solo incorporamos un elemento **<audio>** y sus fuentes correspondientes, dejando el resto del código intacto, incluyendo los archivos externos. No necesitamos cambiar nada más, los eventos, métodos y propiedades son los mismos para los dos medios (audio y video).

> **Hágalo usted mismo:** Cree un nuevo archivo llamado **audio.html**, copie el código del Listado 5-13 dentro de este archivo y ábralo en su navegador. Use los mismos archivos **reproductor.css** y **reproductor.js** creados anteriormente para hacer funcionar su reproductor de audio.

5.6 Referencia rápida

Video y audio son parte esencial de la web. HTML5 incorpora todos los elementos necesarios para aprovechar estas herramientas y utilizarlas en nuestras aplicaciones web.

Elementos

HTML5 provee dos nuevos elementos HTML para procesar medios y una API específica para acceder a la librería de medios.

<video> Este elemento nos permite insertar un archivo de video en un documento HTML.

<audio> Este elemento nos permite insertar un archivo de audio en un documento HTML.

Atributos

La especificación también provee atributos para los elementos `<video>` y `<audio>`:

src Este atributo declara la URL del medio a ser incluido en el documento. Puede usar el elemento `<source>` para proveer más de una fuente y dejar que el navegador elija cual reproducir.

controls Este atributo, si está presente, activa los controles por defecto. Cada navegador provee sus propias funciones, como botones para reproducir y pausar el medio, así como barra de progreso, entre otras.

autoplay Este atributo, si está presente, le indicará al navegador que comience a reproducir el medio lo más pronto posible.

loop Este atributo hará que el navegador reproduzca el medio indefinidamente.

preload Este atributo recomienda al navegador qué hacer con el medio. Puede recibir tres valores diferentes: `none`, `metadata` y `auto`. El valor `none` le dice al navegador que no descargue el archivo hasta que el usuario lo ordene. El valor `metadata` le recomienda al navegador descargar información básica sobre el medio. El valor `auto` le dice al navegador que comience a descargar el archivo tan pronto como sea posible.

Atributos de video

Existen algunos atributos que son específicos para el elemento `<video>`:

poster Este atributo provee una imagen para mostrarla en lugar del video antes de ser reproducido.

width Este atributo determina el tamaño del video en pixeles.

height Este atributo determina el tamaño del video en pixeles.

Eventos

Los eventos más relevantes para esta API son:

progress Este evento es disparado periódicamente para informar el progreso en la descarga del medio.

canplaythrough Este evento es disparado cuando el medio completo puede ser reproducido sin interrupción.

canplay Este evento es disparado cuando el medio puede ser reproducido. A diferencia del evento previo, éste es disparado cuando solo parte del archivo fue descargado (solo los primeros cuadros de un video, por ejemplo).

ended Este evento es disparado cuando la reproducción llega al final del medio.

pause Este evento es disparado cuando la reproducción es pausada.

play Este evento es disparado cuando el medio comienza a ser reproducido.

error Este evento es disparado cuando ocurre un error. El evento es despachado desde el elemento `<source>` (si se encuentra presente) correspondiente a la fuente del medio que produjo el error.

Métodos

Los métodos más comunes para esta API son:

play() Este método comienza o continúa la reproducción del medio.

pause() Este método pausa la reproducción del medio.

load() Este método descarga el archivo del medio. Es útil en aplicaciones dinámicas.

canPlayType(formato) Este método indica si el formato del archivo que queremos utilizar es soportado por el navegador o no. Retorna una cadena vacía si el navegador no puede reproducir el medio y los textos "maybe" (quizás) o "probably" (probablemente) basado en la confianza que tiene de que el medio pueda ser reproducido o no.

Propiedades

Las propiedades más comunes de esta API son:

paused Esta propiedad retorna `true` (verdadero) si la reproducción del medio se encuentra pausada o no ha comenzado.

ended Esta propiedad retorna `true` (verdadero) si la reproducción llegó al final del medio.

duration Esta propiedad retorna la duración del medio en segundos.

currentTime Esta es una propiedad que puede retornar o recibir un valor para informar la posición en la cual el medio se encuentra reproduciendo o establecer una nueva posición donde comenzar a reproducir.

error Esta propiedad retorna el valor del error cuando un error ocurre.

buffered Esta propiedad ofrece información sobre la cantidad del archivo que fue descargado e introducido en el buffer. Retorna un array conteniendo datos sobre cada porción del medio que ha sido descargada. Si el usuario salta a otra parte del medio que no ha sido aún descargada, el navegador comenzará a descargar el medio desde ese punto, generando una nueva porción en el buffer. Los elementos del array son accesibles por medio de los atributos `end()` y `start()`. Por ejemplo, el código `buffered.end(0)` retornará la duración en segundos de la primera porción del medio encontrada en el buffer.

Capítulo 6
Formularios y API Forms

6.1 Formularios Web

La Web 2.0 está completamente enfocada en el usuario. Y cuando el usuario es el centro de atención, todo está relacionado con interfaces, en cómo hacerlas más intuitivas, más naturales, más prácticas, y por supuesto más atractivas. Los formularios web son la interface más importante de todas, permiten a los usuarios insertar datos, tomar decisiones, comunicar información y cambiar el comportamiento de una aplicación. Durante los últimos años, códigos personalizados y librerías fueron creados para procesar formularios en el ordenador del usuario. HTML5 vuelve a estas funciones estándar agregando nuevos atributos, elementos y una API completa. Ahora la capacidad de procesamiento de información insertada en formularios en tiempo real ha sido incorporada en los navegadores y completamente estandarizada.

El elemento <form>

Los formularios en HTML no han cambiado mucho. La estructura sigue siendo la misma, pero HTML5 ha agregado nuevos elementos, tipos de campo y atributos para expandirlos tanto como sea necesario y proveer así las funciones actualmente implementadas en aplicaciones web.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Formularios</title>
</head>
<body>
  <section>
    <form name="miformulario" id="miformulario" method="get">
      <input type="text" name="nombre" id="nombre">
      <input type="submit" value="Enviar">
    </form>
  </section>
</body>
</html>
```

Listado 6-1. *Estructura normal de un formulario.*

En el Listado 6-1 creamos una plantilla básica para formularios. Como puede ver, la estructura del formulario y sus atributos siguen siendo igual que en especificaciones previas. Sin embargo, existen nuevos atributos para el elemento **<form>**:

autocomplete Este es un viejo atributo que se ha vuelto estándar en esta especificación. Puede tomar dos valores: **on** y **off**. El valor por defecto es **on**. Cuando es configurado como **off** los elementos **<input>** pertenecientes a ese formulario tendrán la función de autocompletar desactivada, sin mostrar entradas previas como posibles valores. Puede ser implementado en el elemento **<form>** o en cualquier elemento **<input>** independientemente.

novalidate Una de las características de formularios en HTML5 es la capacidad propia de validación. Los formularios son automáticamente validados. Para evitar este comportamiento, podemos usar el atributo **novalidate**. Para lograr lo mismo para elementos **<input>** específicos, existe otro atributo llamado **formnovalidate**. Ambos atributos son booleanos, ningún valor tiene que ser especificado (su presencia es suficiente para activar su función).

El elemento <input>

El elemento más importante en un formulario es **<input>**. Este elemento puede cambiar sus características gracias al atributo **type** (tipo). Este atributo determina qué clase de entrada es esperada desde el usuario. Los tipos disponibles hasta el momento eran el multipropósitos **text** (para textos en general) y solo unos pocos más específicos como **password** o **submit**. HTML5 ha expandido las opciones incrementando de este modo las posibilidades para este elemento.

En HTML5 estos nuevos tipos no solo están especificando qué clase de entrada es esperada sino también diciéndole al navegador qué debe hacer con la información recibida. El navegador procesará los datos ingresados de acuerdo al valor del atributo **`type`** y validará la entrada o no.

El atributo **`type`** trabaja junto con otros atributos adicionales para ayudar al navegador a limitar y controlar en tiempo real lo ingresado por el usuario.

> **Hágalo usted mismo:** Cree un nuevo archivo HTML con la plantilla del Listado 6-1. Para comprobar cómo funciona cada tipo de campo estudiado de aquí en adelante, reemplace los elementos **`<input>`** en la plantilla por aquellos que quiere probar y abra nuevamente el archivo en su navegador. En este momento la forma en la que los tipos de campo son tratados varía, por este motivo le recomendamos probar el código en cada navegador disponible.

Tipo email

Casi todo formulario en la web ofrece un campo para ingresar una dirección de email, pero hasta ahora el único tipo de campo disponible para esta clase de datos era **`text`**. El tipo **`text`** representa un texto general, no un dato específico, por lo que teníamos que controlar la entrada con código Javascript para estar seguros de que el texto ingresado correspondía a un email válido. Ahora el navegador se hace cargo de esto con el nuevo tipo **`email`**:

```
<input type="email" name="miemail" id="miemail">
```

Listado 6-2. *El tipo `email`.*

El texto insertado en el campo generado por el código del Listado 6-2 será controlado por el navegador y validado como un email. Si la validación falla, el formulario no será enviado.

Cómo cada navegador responderá a una entrada inválida no está determinado en la especificación de HTML5. Por ejemplo, algunos navegadores mostrarán un borde rojo alrededor del elemento **`<input>`** que produjo el error y otros lo mostrarán en azul. Existen formas de personalizar esta respuesta, pero las veremos más adelante.

Tipo search

El tipo **`search`** (búsqueda) no controla la entrada, es solo una indicación para los navegadores. Al detectar este tipo de campo algunos navegadores cambiarán el diseño del elemento para ofrecer al usuario un indicio de su propósito.

```
<input type="search" name="busqueda" id="busqueda">
```

Listado 6-3. *El tipo `search`.*

Tipo url

Este tipo de campo trabaja exactamente igual que el tipo **`email`** pero es específico para direcciones web. Está destinado a recibir solo URLs absolutas y retornará un error si el valor es inválido.

```
<input type="url" name="miurl" id="miurl">
```

Listado 6-4. *El tipo `url`.*

Tipo tel

Este tipo de campo es para números telefónicos. A diferencia de los tipos **`email`** y **`url`**, el tipo **`tel`** no requiere ninguna sintaxis en particular. Es solo una indicación para el navegador en caso de que necesite hacer ajustes de acuerdo al dispositivo en el que la aplicación es ejecutada.

```
<input type="tel" name="telefono" id="telefono">
```

Listado 6-5. *El tipo* `tel`.

Tipo number

Como su nombre lo indica, el tipo **`number`** es sólo válido cuando recibe una entrada numérica. Existen algunos atributos nuevos que pueden ser útiles para este campo:

min El valor de este atributo determina el mínimo valor aceptado para el campo.

max El valor de este atributo determina el máximo valor aceptado para el campo.

step El valor de este atributo determina el tamaño en el que el valor será incrementado o disminuido en cada paso. Por ejemplo, si declara un valor de 5 para **`step`** en un campo que tiene un valor mínimo de 0 y máximo de 10, el navegador no le permitirá especificar valores entre 0 y 5 o entre 5 y 10.

```
<input type="number" name="numero" id="numero" min=”0” max=”10”
                                                         step=”5”>
```

Listado 6-6. *El tipo* `number`.

No es necesario especificar ambos atributos (**`min`** y **`max`**), y el valor por defecto para **`step`** es 1.

Tipo range

Este tipo de campo hace que el navegador construya una nueva clase de control que no existía previamente. Este nuevo control le permite al usuario seleccionar un valor a partir de una serie de valores o rango. Normalmente es mostrado en pantalla como una puntero deslizable o un campo con flechas para seleccionar un valor entre los predeterminados, pero no existe un diseño estándar hasta el momento.

El tipo **`range`** usa los atributos **`min`** y **`max`** estudiados previamente para configurar los límites del rango. También puede utilizar el atributo **`step`** para establecer el tamaño en el cual el valor del campo será incrementado o disminuido en cada paso.

```
<input type="range" name="numero" id="numero" min=”0” max=”10”
                                                        step=”5”>
```

Listado 6-7. *El tipo* `range`.

Podemos declarar el valor inicial utilizando el viejo atributo **`value`** y usar Javascript para mostrar el número seleccionado en pantalla como referencia. Experimentaremos con esto y el nuevo elemento **`<output>`** más adelante.

Tipo date

Este es otro tipo de campo que genera una nueva clase de control. En este caso fue incluido para ofrecer una mejor forma de ingresar una fecha. Algunos navegadores muestran en pantalla un calendario que aparece cada vez que el usuario hace clic sobre el campo. El calendario le permite al usuario seleccionar un día que será ingresado en el campo junto con el resto de la fecha. Un ejemplo de uso es cuando necesitamos proporcionar un método para seleccionar una fecha para un vuelo o la entrada a un espectáculo. Gracias al tipo **`date`** ahora es el navegador el que se encarga de construir un almanaque o las herramientas necesarias para facilitar el ingreso de este tipo de datos.

```
<input type="date" name="fecha" id="fecha">
```

Listado 6-8. *El tipo* `date`.

La interface no fue declarada en la especificación. Cada navegador provee su propia interface y a veces adaptan el diseño al dispositivo en el cual la aplicación está siendo ejecutada. Normalmente el valor generado y esperado tiene la sintaxis **año-mes-día**.

Tipo week

Este tipo de campo ofrece una interface similar a **date**, pero solo para seleccionar una semana completa. Normalmente el valor esperado tiene la sintaxis **2011-W50** donde **2011** es al año y **50** es el número de la semana.

```
<input type="week" name="semana" id="semana">
```

Listado 6-9. *El tipo* `week`.

Tipo month

Similar al tipo de campo previo, éste es específico para seleccionar meses. Normalmente el valor esperado tiene la sintaxis **año-mes**.

```
<input type="month" name="mes" id="mes">
```

Listado 6-10. *El tipo* `month`.

Tipo time

El tipo de campo **time** es similar a **date**, pero solo para la hora. Toma el formato de horas y minutos, pero su comportamiento depende de cada navegador en este momento. Normalmente el valor esperado tiene la sintaxis **hora:minutos:segundos**, pero también puede ser solo **hora:minutos**.

```
<input type="time" name="hora" id="hora">
```

Listado 6-11. *El tipo* `time`.

Tipo datetime

El tipo de campo **datetime** es para ingresar fecha y hora completa, incluyendo la zona horaria.

```
<input type="datetime" name="fechahora" id="fechahora">
```

Listado 6-12. *El tipo* `datetime`.

Tipo datetime-local

El tipo de campo **datetime-local** es como el tipo **datetime** sin la zona horaria.

```
<input type="datetime-local" name="tiempolocal" id="tiempolocal">
```

Listado 6-13. *El tipo* `datetime-local`.

Tipo color

Además de los tipos de campo para fecha y hora existe otro tipo que provee una interface predefinida similar para seleccionar colores. Normalmente el valor esperado para este campo es un número hexadecimal, como #00FF00.

```
<input type="color" name="micolor" id="micolor">
```

Listado 6-14. *El tipo `color`.*

Ninguna interface fue especificada como estándar en HTML5 para el tipo de campo **`color`**, pero es posible que una grilla con un conjunto básico de colores sea adoptada e incorporada en los navegadores.

6.2 Nuevos atributos

Algunos tipos de campo requieren de la ayuda de atributos, como los anteriormente estudiados **min**, **max** y **step**. Otros tipos de campo requieren la asistencia de atributos para mejorar su rendimiento o determinar su importancia en el proceso de validación. Ya vimos algunos de ellos, como **novalidate** para evitar que el formulario completo sea validado o **formnovalidate** para hacer lo mismo con elementos individuales. El atributo **autocomplete**, también estudiado anteriormente, provee medidas de seguridad adicionales para el formulario completo o elementos individuales. Aunque útiles, estos atributos no son los únicos incorporados por HTML5. Es momento de estudiar el resto.

Atributo placeholder

Especialmente en tipos de campo **search**, pero también en entradas de texto normales, el atributo **placeholder** representa una sugerencia corta, una palabra o frase provista para ayudar al usuario a ingresar la información correcta. El valor de este atributo es presentado en pantalla por los navegadores dentro del campo, como una marca de agua que desaparece cuando el elemento es enfocado.

```
<input type="search" name="busqueda" id="busqueda" placeholder="escriba su búsqueda">
```

Listado 6-15. *El atributo `placeholder`.*

Atributo required

Este atributo booleano no dejará que el formulario sea enviado si el campo se encuentra vacío. Por ejemplo, cuando usamos el tipo **email** para recibir una dirección de email, el navegador comprueba si la entrada es un email válido o no, pero validará la entrada si el campo está vacío. Cuando el atributo **required** es incluido, la entrada será válida sólo si se cumplen las dos condiciones: que el campo no esté vacío y que el valor ingresado esté de acuerdo con los requisitos del tipo de campo.

```
<input type="email" name="miemail" id="miemail" required>
```

Listado 6-16. *El campo `email` ahora es un campo requerido.*

Atributo multiple

El atributo **multiple** es otro atributo booleano que puede ser usado en algunos tipos de campo (por ejemplo, **email** o **file**) para permitir el ingreso de entradas múltiples en el mismo campo.

Los valores insertados deben estar separados por coma para ser válidos.

```
<input type="email" name="miemail" id="miemail" multiple>
```

Listado 6-17. *El campo `email` acepta múltiples valores separados por coma.*

El código en el Listado 6-17 permite la inserción de múltiples valores separados por coma, y cada uno de ellos será validado por el navegador como una dirección de email.

Atributo autofocus

Esta es una función que muchos desarrolladores aplicaban anteriormente utilizando el método **focus()** de Javascript. Este método era efectivo pero forzaba el foco sobre el elemento seleccionado, incluso cuando el usuario ya se encontraba posicionado en otro diferente. Este comportamiento era irritante pero difícil de evitar hasta ahora.

El atributo **autofocus** enfocará la página web sobre el elemento seleccionado pero considerando la situación actual. No

moverá el foco cuando ya haya sido establecido por el usuario sobre otro elemento.

```
<input type="search" name="busqueda" id="busqueda" autofocus>
```

Listado 6-18. *El atributo `autofocus` aplicado sobre un campo de búsqueda.*

Atributo pattern

El atributo **`pattern`** es para propósitos de validación. Usa expresiones regulares para personalizar reglas de validación. Algunos de los tipos de campo ya estudiados validan cadenas de texto específicas, pero no permiten hacer validaciones personalizadas, como por ejemplo un código postal que consiste en 5 números. No existe ningún tipo de campo predeterminado para esta clase de entrada.

El atributo **`pattern`** nos permite crear nuestro propio tipo de campo para controlar esta clase de valores no ordinarios. Puede incluso incluir un atributo **`title`** para personalizar mensajes de error.

```
<input pattern=”[0-9]{5}" name="codigopostal" id="codigopostal”
              title=”inserte los 5 números de su código postal”>
```

Listado 6-19. *Tipos personalizados usando el atributo `pattern`.*

IMPORTANTE: Expresiones regulares son un tema complejo y no relacionado directamente con HTML5. Para obtener información adicional al respecto, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Atributo form

El atributo **`form`** es una adición útil que nos permite declarar elementos para un formulario fuera del ámbito de las etiquetas **`<form>`**. Hasta ahora, para construir un formulario teníamos que escribir las etiquetas **`<form>`** de apertura y cierre y luego declarar cada elemento del formulario entre ellas. En HTML5 podemos insertar los elementos en cualquier parte del código y luego hacer referencia al formulario que pertenecen usando su nombre y el atributo **`form`**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Formularios</title>
</head>
<body>
  <nav>
        <input type="search" name="busqueda" id="busqueda"
                                   form="formulario">
  </nav>
  <section>
    <form name="formulario" id="formulario" method="get">
      <input type="text" name="nombre" id="nombre">
      <input type="submit" value="Enviar">
    </form>
  </section>
</body>
</html>
```

Listado 6-20. *Declarando elementos del formulario en cualquier parte.*

6.3 Nuevos elementos para formularios

Ya hemos visto los nuevos tipos de campos disponibles en HTML5, por lo tanto es momento de estudiar los nuevos elementos HTML incorporados con la intención de mejorar o expandir las posibilidades de los formularios.

El elemento <datalist>

El elemento **<datalist>** es un elemento específico de formularios usado para construir una lista de ítems que luego, con la ayuda del atributo **list**, será usada como sugerencia en un campo del formulario.

```
<datalist id="informacion">
  <option value=”123123123” label=”Teléfono 1”>
  <option value=”456456456” label=”Teléfono 2”>
</datalist>
```

Listado 6-21. *Construyendo la lista.*

Este elemento utiliza el elemento **<option>** en su interior para crear la lista de datos a sugerir. Con la lista ya declarada, lo único que resta es referenciarla desde un elemento **<input>** usando el atributo **list**:

```
<input type="tel" name="telefono" id="telefono" list="informacion">
```

Listado 6-22. *Ofreciendo una lista de sugerencias con el atributo* `list`.

El elemento en el Listado 6-22 mostrará posibles valores para que el usuario elija.

IMPORTANTE: El elemento **<datalist>** fue solo implementado en Opera y Firefox Beta en este momento.

El elemento <progress>

Este elemento no es específico de formularios, pero debido a que representa el progreso en la realización de una tarea, y usualmente estas tareas son comenzadas y procesadas a través de formularios, puede ser incluido dentro del grupo de elementos para formularios.

El elemento **<progress>** utiliza dos atributos para configurar su estado y límites. El atributo **value** indica qué parte de la tarea ya ha sido procesada, y **max** declara el valor a alcanzar para que la tarea se considere finalizada. Vamos a usar **<progress>** en futuros ejemplos.

El elemento <meter>

Similar a **<progress>**, el elemento **<meter>** es usado para mostrar una escala, pero no de progreso. Este elemento tiene la intención de representar una medida, como el tráfico del sitio web, por ejemplo.

El elemento **<meter>** cuenta con varios atributos asociados: **min** y **max** configuran los límites de la escala, **value** determina el valor medido, y **low**, **high** y **optimum** son usados para segmentar la escala en secciones diferenciadas y marcar la posición que es óptima.

El elemento <output>

Este elemento representa el resultado de un cálculo. Normalmente ayudará a mostrar los resultados del procesamiento de valores provistos por un formulario. El atributo **for** asocia el elemento **<output>** con el elemento fuente que participa del cálculo, pero este elemento deberá ser referenciado y modificado desde código Javascript. Su sintaxis es **<output>valor</output>**.

6.4 API Forms

Seguramente no le sorprenderá saber que, al igual que cada uno de los aspectos de HTML5, los formularios HTML cuentan con su propia API para personalizar todos los aspectos de procesamiento y validación.

Existen diferentes formas de aprovechar el proceso de validación en HTML5. Podemos usar los tipos de campo para activar el proceso de validación por defecto (por ejemplo, **`email`**) o volver un tipo de campo regular como **`text`** (o cualquier otro) en un campo requerido usando el atributo **`required`**. También podemos crear tipos de campo especiales usando **`pattern`** para personalizar requisitos de validación. Sin embargo, cuando se trata de aplicar mecanismos complejos de validación (por ejemplo, combinando campos o comprobando los resultados de un cálculo) deberemos recurrir a nuevos recursos provistos por esta API.

setCustomValidity()

Los navegadores que soportan HTML5 muestran un mensaje de error cuando el usuario intenta enviar un formulario que contiene un campo inválido.

Podemos crear mensajes para nuestros propios requisitos de validación usando el método **`setCustomValidity(mensaje)`**.

Con este método especificamos un error personalizado que mostrará un mensaje cuando el formulario es enviado. Cuando un mensaje vacío es declarado, el error es anulado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Formularios</title>
  <script>
    function iniciar(){
      nombre1=document.getElementById("nombre");
      nombre2=document.getElementById("apellido");
      nombre1.addEventListener("input", validacion, false);
      nombre2.addEventListener("input", validacion, false);
      validacion();
    }
    function validacion(){
      if(nombre1.value=='' && nombre2.value==''){
        nombre1.setCustomValidity('inserte al menos un nombre');
        nombre1.style.background='#FFDDDD';
      }else{
        nombre1.setCustomValidity('');
        nombre1.style.background='#FFFFFF';
      }
    }
    window.addEventListener("load", iniciar, false);
  </script>
</head>
<body>
  <section>
    <form name="registracion" method="get">
      Nombre:
      <input type="text" name="nombre" id="nombre">
      Apellido:
      <input type="text" name="apellido" id="apellido">
      <input type="submit" id="send" value="ingresar">
    </form>
  </section>
</body>
</html>
```

Listado 6-23. *Declarando errores personalizados.*

El código del Listado 6-23 presenta una situación de validación compleja. Dos campos fueron creados para recibir el nombre y apellido del usuario. Sin embargo, el formulario solo será inválido cuando ambos campos se encuentran vacíos. El usuario necesita ingresar solo uno de los campos, su nombre o su apellido, para validar la entrada.

En casos como éste no es posible usar el atributo **`required`** debido a que no sabemos cuál campo el usuario decidirá utilizar. Solo con código Javascript y errores personalizados podremos crear un efectivo mecanismo de validación para este escenario.

Nuestro código comienza a funcionar cuando el evento **`load`** es disparado. La función **`iniciar()`** es llamada para responder al evento. Esta función crea referencias para los dos elementos **`<input>`** y agrega una escucha para el evento **`input`** en ambos. Estas escuchas llamarán a la función **`validacion()`** cada vez que el usuario escribe dentro de los campos.

Debido a que los elementos **`<input>`** se encuentran vacíos cuando el documento es cargado, debemos declarar una condición inválida para no permitir que el usuario envíe el formulario antes de ingresar al menos uno de los valores. Por esta razón la función **`validacion()`** es llamada al comienzo. Si ambos campos están vacíos el error es generado y el color de fondo del campo **`nombre`** es cambiado a rojo. Sin embargo, si esta condición ya no es verdad porque al menos uno de los campos fue completado, el error es anulado y el color del fondo de **`nombre`** es nuevamente establecido como blanco.

Es importante tener presente que el único cambio producido durante el procesamiento es la modificación del color de fondo del campo. El mensaje declarado para el error con **`setCustomValidity()`** será visible sólo cuando el usuario intente enviar el formulario.

> **Hágalo usted mismo:** Para propósitos de prueba, incluimos el código Javascript dentro del documento. Como resultado, lo único que debe hacer para ejecutar este ejemplo es copiar el código del Listado 6-23 dentro de un archivo HTML vacío y abrir el archivo en su navegador.
>
> **IMPORTANTE:** API Forms está siendo desarrollada en este momento. Depen-diendo del nivel al que la tecnología haya sido adoptada en el momento en el que usted lee estas líneas es probable que necesite probar los códigos de este capítulo en diferentes navegadores para comprobar su correcto funcionamiento.

El evento invalid

Cada vez que el usuario envía el formulario, un evento es disparado si un campo inválido es detectado. El evento es llamado **`invalid`** y es disparado por el elemento que produce el error. Podemos agregar una escucha para este evento y así ofrecer una respuesta personalizada, como en el siguiente ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Formularios</title>
  <script>
    function iniciar(){
      edad=document.getElementById("miedad");
      edad.addEventListener("change", cambiarrango, false);
      document.informacion.addEventListener("invalid",
                                           validacion, true);
      document.getElementById("enviar").addEventListener("click",
                                              enviar, false);
    }
    function cambiarrango(){
      var salida=document.getElementById("rango");
      var calc=edad.value-20;
      if(calc<20){
        calc=0;
        edad.value=20;
      }
      salida.innerHTML=calc+' a '+edad.value;
    }
    function validacion(e){
      var elemento=e.target;
      elemento.style.background='#FFDDDD';
    }
    function enviar(){
      var valido=document.informacion.checkValidity();
```

```
        if(valido){
          document.informacion.submit();
        }
      }
      window.addEventListener("load", iniciar, false);
    </script>
  </head>
  <body>
    <section>
      <form name="informacion" method="get">
        Usuario:
        <input pattern="[A-Za-z]{3,}" name="usuario" id="usuario"
                                        maxlength="10" required>
        Email:
        <input type="email" name="miemail" id="miemail" required>
        Rango de Edad:
        <input type="range" name="miedad" id="miedad" min="0"
                                  max="80" step="20" value="20">
        <output id="rango">0 a 20</output>
        <input type="button" id="enviar" value="ingresar">
      </form>
    </section>
  </body>
  </html>
```

Listado 6-24. *Nuestro propio sistema de validación.*

En el Listado 6-24, creamos un nuevo formulario con tres campos para ingresar el nombre de usuario, un email y un rango de 20 años de edad.

El campo **usuario** tiene tres atributos para validación: el atributo **pattern** solo admite el ingreso de un texto de tres caracteres mínimo, desde la A a la Z (mayúsculas o minúsculas), el atributo **maxlength** limita la entrada a 10 caracteres máximo, y el atributo **required** invalida el campo si está vacío. El campo **miemail** cuenta con sus limitaciones naturales debido a su tipo y además no podrá enviarse vacío gracias al atributo **required**. El campo **miedad** usa los atributos **min**, **max**, **step** y **value** para configurar las condiciones del rango.

También declaramos un elemento **<output>** para mostrar en pantalla una referencia del rango seleccionado.

Lo que el código Javascript hace con este formulario es simple: cuando el usuario hace clic en el botón “ingresar”, un evento **invalid** será disparado desde cada campo inválido y el color de fondo de esos campos será cambiado a rojo por la función **validacion()**.

Veamos este procedimiento con un poco más de detalle. El código comienza a funcionar cuando el típico evento **load** es disparado luego que el documento fue completamente cargado. La función **iniciar()** es ejecutada y tres escuchas son agregadas para los eventos **change**, **invalid** y **click**.

Cada vez que el contenido de los elementos del formulario cambia por alguna razón, el evento **change** es disparado desde ese elemento en particular. Lo que hicimos fue escuchar a este evento desde el campo **range** y llamar a la función **cambiarrango()** cada vez que el evento ocurre. Por este motivo, cuando el usuario desplaza el control del rango o cambia los valores dentro de este campo para seleccionar un rango de edad diferente, los nuevos valores son calculados por la función **cambiarrango()**. Los valores admitidos para este campo son períodos de 20 años (por ejemplo, 0 a 20 o 20 a 40). Sin embargo, el campo solo retorna un valor, como 20, 40, 60 u 80. Para calcular el valor de comienzo del rango, restamos 20 al valor actual del campo con la fórmula **edad.value - 20**, y grabamos el resultado en la variable **calc**. El período mínimo admitido es 0 a 20, por lo tanto con un condicional **if** controlamos esta condición y no permitimos un período menor (estudie la función **cambiarrango()** para entender cómo funciona).

La segunda escucha agregada en la función **iniciar()** es para el evento **invalid**. La función **validacion()** es llamada cuando este evento es disparado para cambiar el color de fondo de los campos inválidos. Recuerde que este evento será disparado desde un campo inválido cuando el botón “ingresar” sea presionado. El evento no contiene una referencia del formulario o del botón “ingresar”, sino del campo que generó el error. En la función **validacion()** esta referencia es capturada y grabada en la variable **elemento** usando la variable **e** y la propiedad **target**. La construcción **e.target** retorna una referencia al elemento **<input>** inválido. Usando esta referencia, en la siguiente línea cambiamos el color de fondo del elemento.

Volviendo a la función **iniciar()**, encontraremos una escucha más que necesitamos analizar. Para tener control absoluto sobre el envío del formulario y el momento de validación, creamos un botón regular en lugar del típico botón **submit**. Cuando este botón es presionado, el formulario es enviado, pero solo si todos sus elementos son válidos. La escucha agregada para el evento **click** en la función **iniciar()** ejecutará la función **enviar()** cuando el usuario haga clic sobre el

botón. Usando el método **checkValidity()** solicitamos al navegador que realice el proceso de validación y solo enviamos el formulario usando el tradicional método **submit()** cuando ya no hay más condiciones inválidas.

Lo que hicimos con el código Javascript fue tomar control sobre todo el proceso de validación, personalizando cada aspecto y modificando el comportamiento del navegador.

Validación en tiempo real

Cuando abrimos el archivo con la plantilla del Listado 6-24 en el navegador, podremos notar que no existe una validación en tiempo real. Los campos son sólo validados cuando el botón "ingresar" es presionado. Para hacer más práctico nuestro sistema personalizado de validación, tenemos que aprovechar los atributos provistos por el objeto **ValidityState**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Formularios</title>
  <script>
    function iniciar(){
      edad=document.getElementById("miedad");
      edad.addEventListener("change", cambiarrango, false);
      document.informacion.addEventListener("invalid",
       validacion, true);
      document.getElementById("enviar").addEventListener("click",
                                              enviar, false);
      document.informacion.addEventListener("input", controlar,
                                                        false);
    }
    function cambiarrango(){
      var salida=document.getElementById("rango");
      var calc=edad.value-20;
      if(calc<20){
        calc=0;
        edad.value=20;
      }
      salida.innerHTML=calc+' a '+edad.value;
    }
    function validacion(e){
      var elemento=e.target;
      elemento.style.background='#FFDDDD';
    }
    function enviar(){
      var valido=document.informacion.checkValidity();
      if(valido){
        document.informacion.submit();
      }
    }
    function controlar(e){
      var elemento=e.target;
      if(elemento.validity.valid){
        elemento.style.background='#FFFFFF';
      }else{
        elemento.style.background='#FFDDDD';
      }
    }
    window.addEventListener("load", iniciar, false);
  </script>
</head>
<body>
  <section>
    <form name="informacion" method="get">
      Usuario:
      <input pattern="[A-Za-z]{3,}" name="usuario" id="usuario"
                                    maxlength="10" required>
      Email:
```

```
      <input type="email" name="miemail" id="miemail" required>
      Rango de Edad:
      <input type="range" name="miedad" id="miedad" min="0"
                                 max="80" step="20" value="20">
      <output id="rango">0 a 20</output>
      <input type="button" id="enviar" value="ingresar">
    </form>
  </section>
</body>
</html>
```

Listado 6-25. *Validando en tiempo real.*

En el Listado 6-25, una nueva escucha fue agregada para el evento **`input`** sobre el formulario. Cada vez que el usuario modifica un campo, escribiendo o cambiando su contenido, la función **`controlar()`** es ejecutada para responder a este evento.

La función **`controlar()`** también aprovecha la propiedad **`target`** para crear una referencia hacia el elemento que disparó el evento **`input`**. La validez del campo es controlada por medio del estado **`valid`** provisto por el atributo **`validity`** en la construcción **`elemento.validity.valid`**.

El estado **`valid`** será **`true`** (verdadero) si el elemento es válido y **`false`** (falso) si no lo es. Usando esta información cambiamos el color del fondo del elemento. Este color será, por lo tanto, blanco para un campo válido y rojo para uno inválido.

Con esta simple incorporación logramos que cada vez que el usuario modifica el valor de un campo del formulario, este campo será controlado y validado, y su condición será mostrada en pantalla en tiempo real.

Propiedades de validación

En el ejemplo del Listado 6-25 controlamos el estado **`valid`**. Este estado particular es un atributo del objeto **`ValidityState`** que retornará el estado de un elemento considerando cada uno de los posibles estados de validación. Si cada condición es válida entonces el valor del atributo **`valid`** será **`true`** (verdadero).

Existen ocho posibles estados de validez para las diferentes condiciones:

valueMissing Este estado es **`true`** (verdadero) cuando el atributo **`required`** fue declarado y el campo está vacío.

typeMismatch Este estado es **`true`** (verdadero) cuando la sintaxis de la entrada no corresponde con el tipo especificado (por ejemplo, si el texto insertado en un tipo de campo **`email`** no es una dirección de email válida).

patternMismatch Este estado es **`true`** (verdadero) cuando la entrada no corresponde con el patrón provisto por el atributo **`pattern`**.

tooLong Este estado es **`true`** (verdadero) cuando el atributo **`maxlength`** fue declarado y la entrada es más extensa que el valor especificado para este atributo.

rangeUnderflow Este estado es **`true`** (verdadero) cuando el atributo **`min`** fue declarado y la entrada es menor que el valor especificado para este atributo.

rangeOverflow Esta estado es **`true`** (verdadero) cuando el atributo **`max`** fue declarado y la entrada es más grande que el valor especificado para este atributo.

stepMismatch Este estado es **`true`** (verdadero) cuando el atributo **`step`** fue declarado y su valor no corresponde con los valores de atributos como **`min`**, **`max`** y **`value`**.

customError Este estado es **`true`** (verdadero) cuando declaramos un error personalizado usando el método **`setCustomValidity()`** estudiado anteriormente.

Para controlar estos estados de validación, debemos utilizar la sintaxis **`elemento.validity.estado`** (donde **`estado`** es cualquiera de los valores listados arriba). Podemos aprovechar estos atributos para saber exactamente que originó el error en un formulario, como en el siguiente ejemplo:

```
function enviar(){
  var elemento=document.getElementById("usuario");
  var valido=document.informacion.checkValidity();

  if(valido){
```

```
      document.informacion.submit();
    }else if(elemento.validity.patternMismatch ||
                                    elemento.validity.valueMissing){
      alert('el nombre de usuario debe tener mínimo de 3 caracteres');
    }
  }
```

Listado 6-26. *Usando estados de validación para mostrar un mensaje de error personalizado.*

En el Listado 6-26, la función **`enviar()`** fue modificada para incorporar esta clase de control. El formulario es validado por el método **`checkValidity()`** y si es válido es enviado con **`submit()`**. En caso contrario, los estados de validación **`patternMismatch`** y **`valueMissing`** para el campo **`usuario`** son controlados y un mensaje de error es mostrado cuando el valor de alguno de ellos es **`true`** (verdadero).

Hágalo usted mismo: Reemplace la función **`enviar()`** en la plantilla del Listado 6-25 con la nueva función del Listado 6-26 y abra el archivo HTML en su navegador.

willValidate

En aplicaciones dinámicas es posible que los elementos involucrados no tengan que ser validados. Este puede ser el caso, por ejemplo, con botones, campos ocultos o elementos como **`<output>`**. La API nos ofrece la posibilidad de detectar esta condición usando el atributo **`willValidate`** y la sintaxis **`elemento.willValidate`**.

6.5 Referencia rápida

Los formularios constituyen el principal medio de comunicación entre usuarios y aplicaciones web. HTML5 incorpora nuevos tipos para el elemento **`<input>`**, una API completa para validar y procesar formularios, y atributos para mejorar esta interface.

Tipos

Algunos de los nuevos tipos de campo introducidos por HTML5 tienen condiciones de validación implícitas. Otros solo declaran un propósito para el campo que ayudará a los navegadores a presentar el formulario en pantalla.

email Este tipo de campo valida la entrada como una dirección de email.

search Este tipo de campo da información al navegador sobre el propósito del campo (búsqueda) para ayudar a presentar el formulario en pantalla.

url Este tipo de campo valida la entrada como una dirección web.

tel Este tipo de campo da información al navegador sobre el propósito del campo (número telefónico) para ayudar a presentar el formulario en pantalla.

number Este tipo de campo valida la entrada como un número. Puede ser combinado con otros atributos (como **`min`**, **`max`** y **`step)`** para limitar los números permitidos.

range Este tipo de campo genera un nuevo control en pantalla para la selección de números. La entrada es limitada por los atributos **`min`**, **`max`** y **`step`**. El atributo **`value`** establece el valor inicial para el elemento.

date Este tipo de campo valida la entrada como una fecha en el formato **`año-mes-día`**.

month Este tipo de campo valida la entrada como una fecha en el formato **`año-mes`**.

week Este tipo de campo valida la entrada como una fecha en el formato **`año-semana`** donde el segundo valor es representado por una letra W y el número de la semana.

time Este tipo de campo valida la entrada como una hora en el formato **`hora:minutos:segundos`**. También puede tomar otras sintaxis como **`hora:minutos`**.

datetime Este tipo de campo valida la entrada como fecha y hora completa, incluyendo zona horaria.

datetime-local Este tipo de campo valida la entrada como una fecha y hora completa, sin zona horaria.

color Este tipo de campo valida la entrada como un valor de color.

Atributos

Nuevos atributos fueron también agregados en HTML5 para mejorar la capacidad de los formularios y ayudar al control de validación.

autocomplete Este atributo especifica si los valores insertados serán almacenados para futura referencia. Puede tomar dos valores: **`on`** y **`off`**.

autofocus Este es un atributo booleano que enfoca el elemento en el que se encuentra cuando la página es cargada.

novalidate Este atributo es exclusivo para elementos **`<form>`**. Es un atributo booleano que establece si el formulario será validado por el navegador o no.

formnovalidate Este atributo es exclusivo para elementos de formulario individuales. Es un atributo booleano que establece si el elemento será validado por el navegador o no.

placeholder Este atributo ofrece información que orientará al usuario sobre la entrada esperada. Su valor puede ser una palabra simple o un texto corto, y será mostrado como una marca de agua dentro del campo hasta que el elemento es enfocado.

required Este atributo declara al elemento como requerido para validación. Es un atributo booleano que no dejará que el formulario sea enviado hasta que una entrada para el campo sea provista.

pattern Este atributo especifica una expresión regular contra la cual la entrada será validada.

multiple Este es un atributo booleano que permite ingresar múltiples valores en el mismo campo (como múltiples cuentas de email, por ejemplo). Los valores deben ser separados por coma.

form Este atributo asocia el elemento al formulario. El valor provisto debe ser el valor del atributo `id` del elemento `<form>`.

list Este atributo asocia el elemento con un elemento `<datalist>` para mostrar una lista de posibles valores para el campo. El valor provisto debe ser el valor del atributo `id` del elemento `<datalist>`.

Elementos

HTML5 también incluye nuevos elementos que ayudan a mejorar y expandir formularios.

<datalist> Este elemento hace posible incluir una lista de opciones predefinidas que será mostrada en un elemento `<input>` como valores sugeridos. La lista es construida con el elemento `<option>` y cada opción es declarada con los atributos `value` y `label`. Esta lista de opciones se relaciona con un elemento `<input>` por medio del atributo `list`.

<progress> Este elemento representa el estado en la evolución de una tarea (por ejemplo, una descarga).

<meter> Este elemento representa una medida, como el tráfico de un sitio web.

<output> Este elemento presenta un valor de salida para aplicaciones dinámicas.

Métodos

HTML5 incluye una API específica para formularios que provee métodos, eventos y propiedades. Algunos de los métodos son:

setCustomValidity(mensaje) Este método nos permite declarar un error y proveer un mensaje de error para un proceso de validación personalizado. Para anular el error, debemos llamar al método con una cadena de texto vacía como atributo.

checkValidity() Este método solicita al navegador iniciar el proceso de validación. Activa el proceso de validación provisto por el navegador sin la necesidad de enviar el formulario. Este método retorna `true` (verdadero) si el elemento es válido.

Eventos

Los eventos incorporados para esta API son los siguientes:

invalid Este evento es disparado cuando un elemento inválido es detectado durante el proceso de validación.

forminput Este evento es disparado cuando un formulario recibe la entrada del usuario.

formchange Este evento es disparado cuando un cambio ocurre en el formulario.

Estado

API Forms provee un grupo de atributos para controlar estados en un proceso de validación personalizado.

valid Este estado es un estado de validación general. Retorna `true` (verdadero) cuando ninguno de los estados restantes es `true` (verdadero), lo que significa que el elemento es válido.

valueMissing Este estado es `true` (verdadero) cuando el atributo `required` fue incluido en el elemento y el campo está vacío.

typeMismatch Este estado es `true` (verdadero) cuando la entrada no es el valor esperado de acuerdo al tipo de campo (por ejemplo, cuando se espera un email o una URL).

patternMismatch Este estado es `true` (verdadero) cuando la entrada no es un valor admitido por la expresión regular especificada con el atributo `pattern`.

tooLong Este estado es `true` (verdadero) cuando el largo de la entrada es mayor que el valor especificado en el atributo `maxlength`.

rangeUnderflow Este estado es `true` (verdadero) cuando la entrada es menor que el valor declarado para el atributo `min`.

rangeOverflow Este estado es `true` (verdadero) cuando la entrada es mayor que el valor declarado para el atributo `max`.

stepMismatch Este estado es `true` (verdadero) cuando el valor declarado para el atributo `step` no corresponde con los valores en los atributos `min`, `max` y `value`.

customError Este estado es `true` (verdadero) cuando un error personalizado fue declarado para el elemento.

Capítulo 7
API Canvas

7.1 Preparando el lienzo

Esta API ofrece una de las más poderosas características de HTML5. Permite a desarrolladores trabajar con un medio visual e interactivo para proveer capacidades de aplicaciones de escritorio para la web.

Al comienzo del libro hablamos sobre cómo HTML5 está reemplazando previos complementos o plug-ins, como Flash o Java applets, por ejemplo. Había dos cosas importantes a considerar para independizar a la web de tecnologías desarrolladas por terceros: procesamiento de video y aplicaciones gráficas. El elemento **<video>** y la API para medios cubren el primer aspecto muy bien, pero no hacen nada acerca de los gráficos. La API Canvas se hace cargo del aspecto gráfico y lo hace de una forma extremadamente efectiva. Canvas nos permite dibujar, presentar gráficos en pantalla, animar y procesar imágenes y texto, y trabaja junto con el resto de la especificación para crear aplicaciones completas e incluso video juegos en 2 y 3 dimensiones para la web.

El elemento <canvas>

Este elemento genera un espacio rectangular vacío en la página web (lienzo) en el cual serán mostrados los resultados de ejecutar los métodos provistos por la API. Cuando es creado, produce sólo un espacio en blanco, como un elemento **<div>** vacío, pero con un propósito totalmente diferente.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Canvas API</title>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="cajalienzo">
    <canvas id="lienzo" width="500" height="300">
      Su navegador no soporta el elemento canvas
    </canvas>
  </section>
</body>
</html>
```

Listado 7-1. *Sintaxis del elemento <canvas>.*

Solo es necesario especificar unos pocos atributos para este elemento, como puede ver en el Listado 7-1. Los atributos **width** (ancho) y **height** (alto) declaran el tamaño del lienzo en pixeles. Estos atributos son necesarios debido a que todo lo que sea dibujado sobre el elemento tendrá esos valores como referencia. Al atributo **id**, como en otros casos, nos facilita el acceso al elemento desde el código Javascript.

Eso es básicamente todo lo que el elemento **<canvas>** hace. Simplemente crea una caja vacía en la pantalla. Es solo a través de Javascript y los nuevos métodos y propiedades introducidos por la API que esta superficie se transforma en algo práctico.

> **IMPORTANTE:** Por razones de compatibilidad, en caso de que Canvas API no se encuentre disponible en el navegador, el contenido entre las etiquetas **<canvas>** será mostrado en pantalla.

getContext()

El método **getContext()** es el primer método que tenemos que llamar para dejar al elemento **<canvas>** listo para trabajar. Genera un contexto de dibujo que será asignado al lienzo. A través de la referencia que retorna podremos aplicar el resto de la API.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
}
window.addEventListener("load", iniciar, false);
```

Listado 7-2. *Creando el contexto de dibujo para el lienzo.*

En el Listado 7-2, una referencia al elemento **<canvas>** fue almacenada en la variable **elemento** y el contexto de dibujo fue creado por **getContext('2d')**. El método puede tomar dos valores: **2d** y **3d**. Esto es, por supuesto, para ambientes de 2 dimensiones y 3 dimensiones. Por el momento solo el contexto **2d** está disponible, pero serios esfuerzos están siendo volcados en el desarrollo de una API estable en 3 dimensiones.

El contexto de dibujo del lienzo será una grilla de pixeles listados en filas y columnas de arriba a abajo e izquierda a derecha, con su origen (el pixel 0,0) ubicado en la esquina superior izquierda del lienzo.

Hágalo usted mismo: Copie el documento HTML del Listado 7-1 dentro de un nuevo archivo vacío. También necesitará crear un archivo llamado **canvas.js** y copiar el código del Listado 7-2 en su interior. Cada código presentado en este capítulo es independiente y reemplaza al anterior.

Conceptos básicos: Cuando una variable es declarada dentro de una función sin la palabra clave **var**, será global. Esto significa que la variable será accesible desde otras partes del código, incluido el interior de funciones. En el código del Listado 7-2, declaramos la variable **lienzo** como global para poder tener siempre acceso al contexto del lienzo.

7.2 Dibujando en el lienzo

Luego de que el elemento **<canvas>** y su contexto han sido inicializados podemos finalmente comenzar a crear y manipular gráficos. La lista de herramientas provista por la API para este propósito es extensa, desde la creación de simples formas y métodos de dibujo hasta texto, sombras o transformaciones complejas. Vamos a estudiarlas una por una.

Dibujando rectángulos

Normalmente el desarrollador deberá preparar la figura a ser dibujada en el contexto (como veremos pronto), pero existen algunos métodos que nos permiten dibujar directamente en el lienzo, sin preparación previa. Estos métodos son específicos para formas rectangulares y son los únicos que generan una forma primitiva (para obtener otras formas tendremos que combinar otras técnicas de dibujo y trazados complejos). Los métodos disponibles son los siguientes:

fillRect(x, y, ancho, alto) Este método dibuja un rectángulo sólido. La esquina superior izquierda será ubicada en la posición especificada por los atributos **x** e **y**. Los atributos **ancho** y **alto** declaran el tamaño.

strokeRect(x, y, ancho, alto) Similar al método anterior, éste dibujará un rectángulo vacío (solo su contorno).

clearRect(x, y, ancho, alto) Esta método es usado para substraer pixeles del área especificada por sus atributos. Es un borrador rectangular.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.strokeRect(100,100,120,120);
  lienzo.fillRect(110,110,100,100);
  lienzo.clearRect(120,120,80,80);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-3. *Dibujando rectángulos.*

Esta es la misma función del Listado 7-2, pero incorpora los nuevos métodos estudiados para dibujar una figura en el lienzo. Como puede ver, el contexto fue asignado a la variable global **lienzo**, y ahora esta variable es usada para referenciar el contexto en cada método.

El primer método usado en la función, **strokeRect(100,100,120,120)**, dibuja un rectángulo vacío con la esquina superior izquierda en la posición **100,100** y un tamaño de 120 pixeles (este es un cuadrado de 120 pixeles). El segundo método, **fillRect(110,110, 100,100)**, dibuja un rectángulo sólido, esta vez comenzando desde la posición **110,110** del lienzo. Y finalmente, con el último método, **clearRect(120,120,80,80)**, un recuadro de 80 pixeles es substraído del centro de la figura.

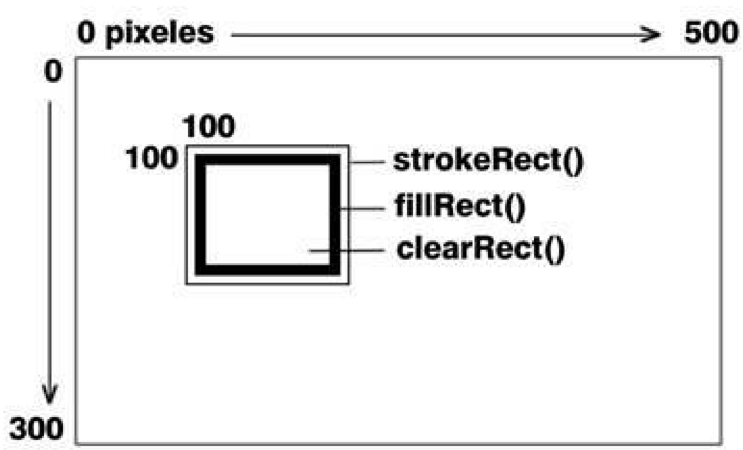


Figura 7-1. *Representación del lienzo y los rectángulos dibujados por el código del Listado 7-3.*

La Figura 7-1 es solo una representación de lo que verá en la pantalla luego de ejecutar el código del Listado 7-3. El elemento **<canvas>** es como una grilla, con su origen en la esquina superior izquierda y el tamaño especificado en sus atributos. Los rectángulos son dibujados en el lienzo en la posición declarada por los atributos **x** e **y**, y uno sobre el otro de acuerdo al orden en el código (el primero en aparecer en el código será dibujado primero, el segundo será dibujado por encima del anterior, y así sucesivamente). Existe un método para personalizar cómo las figuras son dibujadas en pantalla, pero lo veremos más adelante.

Colores

Hasta el momento hemos usado el color otorgado por defecto, negro sólido, pero podemos especificar el color que queremos aplicar mediante sintaxis CSS utilizando las siguientes propiedades:

strokeStyle Esta propiedad declara el color para el contorno de la figura.

fillStyle Esta propiedad declara el color para el interior de la figura.

globalAlpha Esta propiedad no es para definir color sino transparencia. Especifica la transparencia para todas las figuras dibujadas en el lienzo.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  lienzo.fillStyle="#000099";
  lienzo.strokeStyle="#990000";

  lienzo.strokeRect(100,100,120,120);
  lienzo.fillRect(110,110,100,100);
  lienzo.clearRect(120,120,80,80);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-4. *Aplicando color.*

Los colores en el Listado 7-4 fueron declarados usando números hexadecimales. Podemos también usar funciones como **rgb()** o incluso especificar transparencia para la figura aprovechando la función **rgba()**. Estos métodos deben ser siempre escritos entre comillas (por ejemplo, **strokeStyle="rgba(255,165,0,1)"**).

Cuando un nuevo color es especificado se vuelve el color por defecto para el resto de los dibujos, a menos que volvamos a cambiarlo más adelante.

A pesar de que el uso de la función **rgba()** es posible, existe otra propiedad más específica para declarar el nivel de transparencia: **globalAlpha**. Su sintaxis es **globalAlpha=valor**, donde **valor** es un número entre 0.0 (totalmente opaco) y 1.0 (totalmente transparente).

Gradientes

Gradientes son una herramienta esencial en cualquier programa de dibujo estos días, y esta API no es la excepción. Así como en CSS3, los gradientes en la API Canvas pueden ser lineales o radiales, y pueden incluir puntos de terminación para combinar colores.

createLinearGradient(x1, y1, x2, y2) Este método crea un objeto que luego será usado para aplicar un gradiente lineal al lienzo.

createRadialGradient(x1, y1, r1, x2, y2, r2) Este método crea un objeto que luego será usado para aplicar un gradiente circular o radial al lienzo usando dos círculos. Los valores representan la posición del centro de cada círculo y sus radios.

addColorStop(posición, color) Este método especifica los colores a ser usados por el gradiente. El atributo **posición** es un valor entre 0.0 y 1.0 que determina dónde la degradación comenzará para ese color en particular.

```
function iniciar(){
```

```
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var gradiente=lienzo.createLinearGradient(0,0,10,100);
  gradiente.addColorStop(0.5, '#0000FF');
  gradiente.addColorStop(1, '#000000');
  lienzo.fillStyle=gradiente;

  lienzo.fillRect(10,10,100,100);
  lienzo.fillRect(150,10,200,100);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-5. *Aplicando un gradiente lineal al lienzo.*

En el Listado 7-5, creamos el objeto gradiente desde la posición **0,0** a la **10,100**, otorgando una leve inclinación hacia la izquierda. Los colores fueron declarados por el método **addColorStop()** y el gradiente logrado fue finalmente aplicado a la propiedad **fillStyle**, como un color regular.

Note que las posiciones del gradiente son correspondientes al lienzo, no a las figuras que queremos afectar. El resultado es que si movemos los rectángulos dibujados al final de la función hacia una nueva posición, el gradiente para esos triángulos cambiará.

Hágalo usted mismo: El gradiente radial es similar al de CSS3. Intente reemplazar el gradiente lineal en el código del Listado 7-5 por un gradiente radial usando una expresión como **createRadialGradient(0,0,30,0,0,300)**. También puede experimentar moviendo los rectángulos de posición para ver cómo el gradiente es aplicado a estas figuras.

Creando trazados

Los métodos estudiados hasta el momento dibujan directamente en el lienzo, pero ese no es siempre el caso. Normalmente tendremos que procesar figuras en segundo plano y una vez que el trabajo esté hecho enviar el resultado al contexto para que sea dibujado. Con este propósito, API Canvas introduce varios métodos con los que podremos generar trazados.

Un trazado es como un mapa a ser seguido por el lápiz. Una vez declarado, el trazado será enviado al contexto y dibujado de forma permanente en el lienzo. El trazado puede incluir diferentes tipos de líneas, como líneas rectas, arcos, rectángulos, entre otros, para crear figuras complejas.

Existen dos métodos para comenzar y cerrar el trazado:

beginPath() Este método comienza la descripción de una nueva figura. Es llamado en primer lugar, antes de comenzar a crear el trazado.

closePath() Este método cierra el trazado generando una línea recta desde el último punto hasta el punto de origen. Puede ser ignorado cuando utilizamos el método **fill()** para dibujar el trazado en el lienzo.

También contamos con tres métodos para dibujar el trazado en el lienzo:

stroke() Este método dibuja el trazado como una figura vacía (solo el contorno).

fill() Este método dibuja el trazado como una figura sólida. Cuando usamos este método no necesitamos cerrar el trazado con **closePath()**, el trazado es automáticamente cerrado con una línea recta trazada desde el punto final hasta el origen.

clip() Este método declara una nueva área de corte para el contexto. Cuando el contexto es inicializado, el área de corte es el área completa ocupada por el lienzo. El método **clip()** cambiará el área de corte a una nueva forma creando de este modo una máscara. Todo lo que caiga fuera de esa máscara no será dibujado.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  // aquí va el trazado
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-6. *Reglas básicas para trabajar con trazados.*

El código del Listado 7-6 no crea absolutamente nada, solo incorpora los métodos necesarios para iniciar y luego dibujar el trazado en pantalla. Para crear el trazado y la figura real que será enviada al contexto y dibujada en el lienzo, contamos con varios métodos disponibles:

moveTo(x, y) Este método mueve el lápiz a una posición específica para continuar con el trazado. Nos permite comenzar o continuar el trazado desde diferentes puntos, evitando líneas continuas.

lineTo(x, y) Este método genera una línea recta desde la posición actual del lápiz hasta la nueva declarada por los atributos **x** e **y**.

rect(x, y, ancho, alto) Este método genera un rectángulo. A diferencia de los métodos estudiados anteriormente, éste generará un rectángulo que formará parte del trazado (no directamente dibujado en el lienzo). Los atributos tienen la misma función.

arc(x, y, radio, ángulo inicio, ángulo final, dirección) Este método genera un arco o un círculo en la posición **x** e **y**, con un radio y desde un ángulo declarado por sus atributos. El último valor es un valor booleano (falso o verdadero) para indicar la dirección a favor o en contra de las agujas del reloj.

quadraticCurveTo(cpx, cpy, x, y) Este método genera una curva Bézier cuadrática desde la posición actual del lápiz hasta la posición declarada por los atributos **x** e **y**. Los atributos **cpx** y **cpy** indican el punto que dará forma a la curva.

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) Este método es similar al anterior pero agrega dos atributos más para generar una curva Bézier cúbica. Ahora disponemos de dos puntos para moldear la curva, declarados por los atributos **cp1x**, **cp1y**, **cp2x** y **cp2y**.

Veamos un trazado sencillo para entender cómo funcionan:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.moveTo(100,100);
  lienzo.lineTo(200,200);
  lienzo.lineTo(100,200);
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-7. *Nuestro primer trazado.*

Recomendamos siempre establecer la posición inicial del lápiz inmediatamente después de iniciar el trazado con **beginPath()**. En el código del Listado 7-7 el primer paso fue mover el lápiz a la posición **100,100** y luego generar una línea desde ese punto hasta el punto **200,200**. Ahora la posición del lápiz es **200,200** y la siguiente línea será generada desde aquí hasta el punto **100,200**. Finalmente, el trazado es dibujado en el lienzo como una forma vacía con el método **stroke()**.

Si prueba el código en su navegador, verá un triángulo abierto en la pantalla. Este triángulo puede ser cerrado o incluso rellenado y transformado en una figura sólida usando diferentes métodos, como vemos en el siguiente ejemplo:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.moveTo(100,100);
  lienzo.lineTo(200,200);
  lienzo.lineTo(100,200);
  lienzo.closePath();
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-8. *Completando el triángulo.*

El método **closePath()** simplemente agrega una línea recta al trazado, desde el último al primer punto, cerrando la figura.

Usando el método **stroke()** al final de nuestro trazado dibujamos un triángulo vacío en el lienzo. Para lograr una figura sólida, este método debe ser reemplazado por **fill()**:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.moveTo(100,100);
  lienzo.lineTo(200,200);
  lienzo.lineTo(100,200);
  lienzo.fill();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-9. *Triángulo sólido.*

Ahora la figura en la pantalla será un triángulo sólido. El método **fill()** cierra el trazado automáticamente, por lo que ya no tenemos que usar **closePath()** para lograrlo.

Uno de los métodos mencionados anteriormente para dibujar un trazado en el lienzo fue **clip()**. Este método en realidad no dibuja nada, lo que hace es crear una máscara con la forma del trazado para seleccionar qué será dibujado y qué no. Todo lo que caiga fuera de la máscara no se dibujará en el lienzo. Veamos un ejemplo:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  lienzo.beginPath();
  lienzo.moveTo(100,100);
  lienzo.lineTo(200,200);
  lienzo.lineTo(100,200);
  lienzo.clip();

  lienzo.beginPath();
  for(f=0; f<300; f=f+10){
    lienzo.moveTo(0,f);
    lienzo.lineTo(500,f);
  }
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-10. *Usando el triángulo anterior como una máscara.*

Para mostrar exactamente cómo funciona el método **clip()**, en el Listado 7-10 utilizamos un bucle **for** para crear líneas horizontales cada 10 pixeles. Estas líneas van desde el lado izquierdo al lado derecho del lienzo, pero solo las partes de las líneas que caen dentro de la máscara (el triángulo) serán dibujadas.

Ahora que ya sabemos cómo dibujar trazados, es tiempo de ver el resto de las alternativas con las que contamos para crearlos. Hasta el momento hemos estudiado cómo generar líneas rectas y formas rectangulares. Para figuras circulares, la API provee tres métodos: **arc()**, **quadraticCurveTo()** y **bezierCurveTo()**. El primero es relativamente sencillo y puede generar círculos parciales o completos, como mostramos en el siguiente ejemplo:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.arc(100,100,50,0,Math.PI*2, false);
  lienzo.stroke();
```

```
}
window.addEventListener("load", iniciar, false);
```

Listado 7-11. *Círculos con* `arc()`.

Lo primero que seguramente notará en el método **`arc()`** en nuestro ejemplo es el uso del valor **`PI`**. Este método usa radianes en lugar de grados para los valores del ángulo. En radianes, el valor **`PI`** representa 180 grados, por lo que la formula **`PI*2`** multiplica **`PI`** por **`2`** obteniendo un ángulo de 360 grados.

El código en el Listado 7-11 genera un arco con centro en el punto **`100,100`** y un radio de 50 pixeles, comenzando a **`0`** grados y terminando a **`Math.PI*2`** grados, lo que representa un círculo completo. El uso de la propiedad **`PI`** del objeto **`Math`** nos permite obtener el valor preciso de PI.

Si necesitamos calcular el valor en radianes de cualquier ángulo en grados usamos la fórmula: **`Math.PI / 180 × grados`**, como en el próximo ejemplo:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  var radianes=Math.PI/180*45;
  lienzo.arc(100,100,50,0,radianes, false);
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-12. *Un arco de 45 grados.*

Con el código del Listado 7-12 obtenemos un arco que cubre 45 grados de un círculo. Intente cambiar el valor de la dirección a **`true`** (verdadero). En este caso, el arco será generado desde 0 grados a 315, creando un círculo abierto.

Una cosa importante a considerar es que si continuamos construyendo el trazado luego del arco, el actual punto de comienzo será el final del arco. Si no deseamos que esto pase tendremos que usar el método **`moveTo()`** para cambiar la posición del lápiz, como hicimos anteriormente. Sin embargo, si la próxima figura es otro arco (por ejemplo, un círculo completo) siempre recuerde que el método **`moveTo()`** mueve el lápiz virtual hacia el punto en el cual el círculo comenzará a ser dibujado, no el centro del círculo. Digamos que el centro del círculo que queremos dibujar se encuentra en el punto **`300,150`** y su radio es de **`50`**. El método **`moveTo()`** debería mover el lápiz a la posición **`350,150`** para comenzar a dibujar el círculo.

Además de **`arc()`**, existen dos métodos más para dibujar curvas, en este caso curvas complejas. El método **`quadraticCurveTo()`** genera una curva Bézier cuadrática, y el método **`bezierCurveTo()`** es para curvas Bézier cúbicas. La diferencia entre estos dos métodos es que el primero cuenta con un solo punto de control y el segundo con dos, creando de este modo diferentes tipos de curvas.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.moveTo(50,50);
  lienzo.quadraticCurveTo(100,125, 50,200);
  lienzo.moveTo(250,50);
  lienzo.bezierCurveTo(200,125, 300,125, 250,200);
  lienzo.stroke();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-13. *Curvas complejas.*

Para la curva cuadrática movimos el lápiz virtual a la posición **`50,50`** y finalizamos la curva en el punto **`50,200`**. El punto de control para esta curva fue ubicado en la posición **`100,125`**.

La curva generada por el método **`bezierCurveTo()`** es un poco más compleja. Hay dos puntos de control para esta curva, el primero en la posición **`200,125`** y el segundo en la posición **`300,125`**.

Los valores en la Figura 7-2 indican los puntos de control para las curvas. Moviendo estos puntos cambiamos la

forma de la curva.

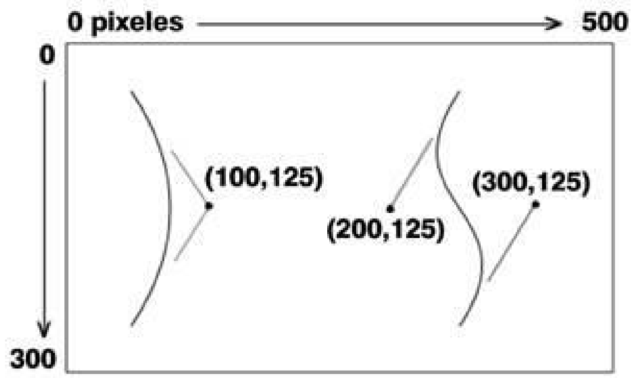


Figura 7-2. *Representación de curvas Bézier y sus puntos de control.*

Hágalo usted mismo: Puede agregar tantas curvas como necesite para construir su figura. Intente cambiar los valores de los puntos de control en el Listado 7-13 para ver cómo afectan a las curvas. Construya figuras más complejas combinando curvas y líneas para entender cómo la construcción del trazado es realizada.

Estilos de línea

Hasta esta parte del capítulo hemos usado siempre los mismos estilos de líneas. El ancho, la terminación y otros aspectos de la línea pueden ser modificados para obtener exactamente el tipo de línea que necesitamos para nuestros dibujos.

Existen cuatro propiedades específicas para este propósito:

lineWidth Esta propiedad determina el grosor de la línea. Por defecto el valor es 1.0 unidades.

lineCap Esta propiedad determina la forma de la terminación de la línea. Puede recibir uno de estos tres valores: **`butt`**, **`round`** y **`square`**.

lineJoin Esta propiedad determina la forma de la conexión entre dos líneas. Los valores posibles son: **`round`**, **`bevel`** y **`miter`**.

miterLimit Trabajando en conjunto con **`lineJoin`**, esta propiedad determina cuánto la conexión de dos líneas será extendida cuando la propiedad **`lineJoin`** es declarada con el valor **`miter`**.

Las propiedades afectarán el trazado completo. Cada vez que tenemos que cambiar las características de las líneas debemos crear un nuevo trazado.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.beginPath();
  lienzo.arc(200,150,50,0,Math.PI*2, false);
  lienzo.stroke();

  lienzo.lineWidth=10;
  lienzo.lineCap="round";
  lienzo.beginPath();
  lienzo.moveTo(230,150);
  lienzo.arc(200,150,30,0,Math.PI, false);
  lienzo.stroke();

  lienzo.lineWidth=5;
  lienzo.lineJoin="miter";
  lienzo.beginPath();
  lienzo.moveTo(195,135);
  lienzo.lineTo(215,155);
  lienzo.lineTo(195,155);
  lienzo.stroke();
```

```
}
window.addEventListener("load", iniciar, false);
```

Listado 7-14. *Propiedades para probar diferentes estilos de línea.*

Comenzamos el dibujo en el código del Listado 7-14 creando un trazado para un círculo completo con propiedades por defecto. Luego, usando **lineWith**, cambiamos el ancho de la línea a 10 y definimos la propiedad **lineCap** como **round**. Esto hará que el siguiente trazado sea más grueso y con terminaciones redondeadas. Para crear un trazado con estas características, primero movimos el lápiz a la posición **230,150** y luego generamos un semicírculo. Los extremos redondeados nos ayudarán a simular una boca sonriente.

Finalmente, agregamos un trazado creado con dos líneas para lograr una forma similar a una nariz. Las líneas para este trazado fueron configuradas con un ancho de **5** y serán unidas de acuerdo a la propiedad **lineJoin** y su valor **miter**. Esta propiedad hará a la nariz lucir puntiaguda, expandiendo las puntas de las líneas en la unión hasta que ambas alcancen un punto en común.

Hágalo usted mismo: Experimente con las líneas para la nariz modificando la propiedad **miterLimit** (por ejemplo, con la instrucción **miterLimit=2**). Cambie el valor de la propiedad **lineJoin** a **round** o **bevel**. También puede modificar la forma de la boca probando diferentes valores para la propiedad **lineCap**.

Texto

Escribir texto en el lienzo es tan simple como definir unas pocas propiedades y llamar al método apropiado. Tres propiedades son ofrecidas para configurar texto:

font Esta propiedad tiene una sintaxis similar a la propiedad **font** de CSS, y acepta los mismos valores.

textAlign Esta propiedad alinea el texto. Existen varios valores posibles: **start** (comienzo), **end** (final), **left** (izquierda), **right** (derecha) y **center** (centro).

textBaseline Esta propiedad es para alineamiento vertical. Establece diferentes posiciones para el texto (incluyendo texto Unicode). Los posibles valores son: **top**, **hanging**, **middle**, **alphabetic**, **ideographic** y **bottom**.

Dos métodos están disponibles para dibujar texto en el lienzo:

strokeText(texto, x, y) Del mismo modo que el método **stroke()** para el trazado, este método dibujará el texto especificado en la posición **x,y** como una figura vacía (solo los contornos). Puede también incluir un cuarto valor para declarar el tamaño máximo. Si el texto es más extenso que este último valor, será encogido para caber dentro del espacio establecido.

fillText(texto, x, y) Este método es similar al método anterior excepto que esta vez el texto dibujado será sólido (igual que la función para el trazado).

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.font="bold 24px verdana, sans-serif";
  lienzo.textAlign="start";
  lienzo.fillText("Mi mensaje", 100,100);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-15. *Dibujando texto.*

Como podemos ver en el Listado 7-15, la propiedad **font** puede tomar varios valores a la vez, usando exactamente la misma sintaxis que CSS. La propiedad **textAling** hace que el texto sea dibujado desde la posición **100,100** (si el valor de esta propiedad fuera **end**, por ejemplo, el texto terminaría en la posición **100,100**). Finalmente, el método **fillText** dibuja un texto sólido en el lienzo.

Además de los previamente mencionados, la API provee otro método importante para trabajar con texto:

measureText() Este método retorna información sobre el tamaño de un texto específico. Puede ser útil para combinar texto con otras formas en el lienzo y calcular posiciones o incluso colisiones en animaciones.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  lienzo.font="bold 24px verdana, sans-serif";
  lienzo.textAlign="start";
  lienzo.textBaseline="bottom";
  lienzo.fillText("Mi mensaje", 100,124);

  var tamano=lienzo.measureText("Mi mensaje");
  lienzo.strokeRect(100,100,tamano.width,24);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-16. *Midiendo texto.*

En este ejemplo comenzamos con el mismo código del Listado 7-15, pero agregamos un alineamiento vertical. La propiedad **`textBaseline`** fue establecida como **`bottom`** (inferior), lo que significa que la base o parte inferior del texto estará ubicada en la posición **`124`**. Esto nos ayudará a conocer la posición vertical exacta del texto en el lienzo.

Usando el método **`measureText()`** y la propiedad **`width`** (ancho) obtenemos el tamaño horizontal del texto. Con esta medida estamos listos para dibujar un rectángulo que rodeará al texto.

> **Hágalo usted mismo:** Utilizando el código del Listado 7-16, pruebe diferentes valores para las propiedades **`textAlign`** y **`textBaseline`**. Use el rectángulo como referencia para comprobar cómo estas propiedades trabajan. Escriba un texto diferente para ver cómo el rectángulo se adapta automáticamente a su tamaño.

Sombras

Por supuesto, sombras son también una parte importante de Canvas API. Podemos generar sombras para cada trazado e incluso textos. La API provee cuatro propiedades para hacerlo:

shadowColor Esta propiedad declara el color de la sombra usando sintaxis CSS.

shadowOffsetX Esta propiedad recibe un número para determinar qué tan lejos la sombra estará ubicada del objeto (dirección horizontal).

shadowOffsetY Esta propiedad recibe un número para determinar qué tan lejos la sombra estará ubicada del objeto (dirección vertical).

shadowBlur Esta propiedad produce un efecto de difuminación para la sombra.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.shadowColor="rgba(0,0,0,0.5)";
  lienzo.shadowOffsetX=4;
  lienzo.shadowOffsetY=4;
  lienzo.shadowBlur=5;
  lienzo.font="bold 50px verdana, sans-serif";
  lienzo.fillText("Mi mensaje ", 100,100);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-17. *Aplicando sombras.*

La sombra creada en el Listado 7-17 usa la función **`rgba()`** para obtener un color negro semitransparente. Es desplazada **`4`** pixeles del objeto y tiene un valor de difuminación de **`5`**.

> **Hágalo usted mismo:** Aplique sombras a otra figura en lugar de texto. Por ejemplo, pruebe generar sombras para figuras vacías y sólidas, usando rectángulos o círculos.

Transformaciones

LAAPI Canvas ofrece operaciones complejas que es posible aplicar sobre el lienzo para afectar los gráficos que luego son dibujados en él. Estas operaciones son realizadas utilizando cinco métodos de transformación diferentes, cada uno para un propósito específico.

translate(x, y) Este método de transformación es usado para mover el origen del lienzo. Cada lienzo comienza en el punto 0,0 localizado en la esquina superior izquierda, y los valores se incrementan en cualquier dirección dentro del lienzo. Valores negativos caen fuera del lienzo. Aveces es bueno poder usar valores negativos para crear figuras complejas. El método **`translate()`** nos permite mover el punto 0,0 a una posición específica para usar el origen como referencia para nuestros dibujos o para aplicar otras transformaciones.

rotate(ángulo) Este método de transformación rotará el lienzo alrededor del origen tantos ángulos como sean especificados.

scale(x, y) Este método de transformación incrementa o disminuye las unidades de la grilla para reducir o ampliar todo lo que esté dibujado en el lienzo. La escala puede ser cambiada independientemente para el valor horizontal o vertical usando los atributos **`x`** e **`y`**. Los valores pueden ser negativos, produciendo un efecto de espejo. Por defecto los valores son iguales a 1.0.

transform(m1, m2, m3, m4, dx, dy) El lienzo contiene una matriz de valores que especifican sus propiedades. El método **`transform()`** aplica una nueva matriz sobre la actual para modificar el lienzo.

setTransform(m1, m2, m3, m4, dx, dy) Este método reinicializa la actual matriz de transformación y establece una nueva desde los valores provistos en sus atributos.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.font="bold 20px verdana, sans-serif";
  lienzo.fillText("PRUEBA",50,20);
  lienzo.translate(50,70);
  lienzo.rotate(Math.PI/180*45);
  lienzo.fillText("PRUEBA",0,0);
  lienzo.rotate(-Math.PI/180*45);
  lienzo.translate(0,100);
  lienzo.scale(2,2);
  lienzo.fillText("PRUEBA",0,0);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-18. *Moviendo, rotando y escalando.*

No hay mejor forma de entender cómo funcionan las transformaciones que usarlas en nuestro código. En el Listado 7-18, aplicamos los métodos **`translate()`**, **`rotate()`** y **`scale()`** al mismo texto. Primero dibujamos un texto en el lienzo con la configuración por defecto. El texto aparecerá en la posición **`50,20`** con un tamaño de 20 pixeles. Luego de esto, usando **`translate()`**, el origen del lienzo es movido a la posición **`50,70`** y el lienzo completo es rotado 45 grados con el método **`rotate()`**. Otro texto es dibujado en el nuevo origen, con una inclinación de 45 grados. Las transformaciones aplicadas se vuelven los valores por defecto, por lo tanto antes de aplicar el siguiente método **`scale()`** rotamos el lienzo 45 grados negativos para ubicarlo en su posición original. Realizamos una transformación más moviendo el origen otros 100 pixeles hacia abajo. Finalmente, la escala del lienzo es duplicada y un nuevo texto es dibujado al doble del tamaño de los anteriores.

Cada transformación es acumulativa. Si realizamos dos transformaciones usando **`scale()`**, por ejemplo, el segundo método realizará el escalado considerando el estado actual del lienzo. Una orden **`scale(2,2)`** luego de otra **`scale(2,2)`** cuadruplicará la escala del lienzo. Y para los métodos de transformación de la matriz, esta no es una excepción. Es por esto que contamos con dos métodos para realizar esta clase de transformaciones: **`transform()`** y **`setTransform()`**.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.transform(3,0,0,1,0,0);
  lienzo.font="bold 20px verdana, sans-serif";
  lienzo.fillText("PRUEBA",20,20);
  lienzo.transform(1,0,0,10,0,0);
  lienzo.font="bold 20px verdana, sans-serif";
```

```
  lienzo.fillText("PRUEBA",100,20);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-19. *Transformaciones acumulativas sobre la matriz.*

Al igual que en el código anterior, en el Listado 7-19 aplicamos varios métodos de transformación sobre el mismo texto para comparar efectos. Los valores por defecto de la matriz del lienzo son **1,0,0,1,0,0**. Cambiando el primer valor a **3**, en la primera transformación de nuestro ejemplo arriba, estiramos el lienzo horizontalmente. El texto dibujado luego de esta transformación será más ancho que en condiciones por defecto.

Con la siguiente transformación, el lienzo fue estirado verticalmente cambiando el cuarto valor a **10** y preservando los anteriores.

Un detalle importante a recordar es que las transformaciones son aplicadas sobre la matriz declarada en previas transformaciones, por lo que el segundo texto mostrado por el código del Listado 7-19 será igual de ancho que el anterior (es estirado horizontal y verticalmente). Para reinicializar la matriz y declarar nuevos valores de transformación, podemos usar el método **setTransform()**.

> **Hágalo usted mismo:** Reemplace el último método **transform()** en el ejemplo por **setTransform()** y compruebe los resultados. Usando solo un texto, cambie cada valor en el método **transform()** para conocer la clase de transformación realizada en el lienzo por cada uno de ellos.

Restaurando el estado

La acumulación de transformaciones hace realmente difícil volver a anteriores estados. En el código del Listado 7-18, por ejemplo, tuvimos que recordar el valor de rotación usado previamente para poder realizar una nueva rotación y volver el lienzo al estado original. Considerando situaciones como ésta, Canvas API provee dos métodos para grabar y recuperar el estado del lienzo.

save() Este método graba el estado del lienzo, incluyendo transformaciones ya aplicadas, valores de propiedades de estilo y la actual máscara (el área creada por el método **clip()**, si existe).

restore() Este método recupera el último estado grabado.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.save();
  lienzo.translate(50,70);
  lienzo.font="bold 20px verdana, sans-serif";
  lienzo.fillText("PRUEBA1",0,30);
  lienzo.restore();
  lienzo.fillText("PRUEBA2",0,30);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-20. *Grabando el estado del lienzo.*

Si ejecuta el código del Listado 7-20 en su navegador, verá el texto “PRUEBA1” en grandes letras al centro del lienzo, y el texto “PRUEBA2” en letras pequeñas, cercano al origen. Lo que hicimos fue grabar el estado por defecto del lienzo y luego establecer una nueva posición para el origen y estilos para el texto. El primer texto es dibujado con esta configuración, pero antes de dibujar el segundo texto el estado original es restaurado, por lo que este texto es mostrado con los estilos por defecto, no con los declarados para el primero.

No importa cuántas transformaciones hayamos realizado, luego de llamar al método **restore()** la configuración del lienzo será retornada exactamente a su estado anterior (el último grabado).

globalCompositeOperation

Cuando hablamos de trazados dijimos que existe una propiedad para determinar cómo una figura es posicionada y combinada con figuras dibujadas previamente en el lienzo. La propiedad es **globalCompositeOperation** y su valor por defecto es **source-over**, lo que significa que la nueva figura será dibujada sobre las que ya existen en el lienzo. La

propiedad ofrece 11 valores más:

source-in Solo la parte de la nueva figura que se sobrepone a las figuras previas es dibujada. El resto de la figura, e incluso el resto de las figuras previas, se vuelven transparentes.

source-out Solo la parte de la nueva figura que no se sobrepone a las figuras previas es dibujada. El resto de la figura, e incluso el resto de las figuras previas, se vuelven transparentes.

source-atop Solo la parte de la nueva figura que se superpone con las figuras previas es dibujada. Las figuras previas son preservadas, pero el resto de la nueva figura se vuelve transparente.

lighter Ambas figuras son dibujadas (nueva y vieja), pero el color de las partes que se superponen es obtenido adicionando los valores de los colores de cada figura.

xor Ambas figuras son dibujadas (nueva y vieja), pero las partes que se superponen se vuelven transparentes.

destination-over Este es el opuesto del valor por defecto. Las nuevas figuras son dibujadas detrás de las viejas que ya se encuentran en el lienzo.

destination-in Las partes de las figuras existentes en el lienzo que se superponen con la nueva figura son preservadas. El resto, incluyendo la nueva figura, se vuelven transparentes.

destination-out Las partes de las figuras existentes en el lienzo que no se superponen con la nueva figura son preservadas. El resto, incluyendo la nueva figura, se vuelven transparentes.

destination-atop Las figuras existentes y la nueva son preservadas solo en la parte en la que se superponen.

darker Ambas figuras son dibujadas, pero el color de las partes que se superponen es determinado substrayendo los valores de los colores de cada figura.

copy Solo la nueva figura es dibujada. Las ya existentes se vuelven transparentes.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  lienzo.fillStyle="#990000";
  lienzo.fillRect(100,100,300,100);
  lienzo.globalCompositeOperation="destination-atop";
  lienzo.fillStyle="#AAAAFF";
  lienzo.font="bold 80px verdana, sans-serif";
  lienzo.textAlign="center";
  lienzo.textBaseline="middle";
  lienzo.fillText("PRUEBA",250,110);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-21. *Probando la propiedad* `globalCompositeOperation`.

Solo representaciones visuales de cada posible valor para la propiedad **`globalCompositeOperation`** le ayudarán a comprender cómo funcionan. Con este propósito, preparamos el código del Listado 7-21. Cuando este código es ejecutado, un rectángulo rojo es dibujado en el medio del lienzo, pero gracias al valor **`destination-atop`** solo la parte del rectángulo que se superpone con el texto es dibujada.

Hágalo usted mismo: Reemplace el valor **`destination-atop`** con cualquiera de los demás valores posibles para esta propiedad y compruebe el resultado en su navegador. Pruebe el código en distintos navegadores.

7.3 Procesando imágenes

API Canvas no sería nada sin la capacidad de procesar imágenes. Pero incluso cuando las imágenes son un elemento tan importante para una aplicación gráfica, solo un método nativo fue provisto para trabajar con ellas.

drawImage()

El método **`drawImage()`** es el único a cargo de dibujar una imagen en el lienzo. Sin embargo, este método puede recibir un número de valores que producen diferentes resultados. Estudiemos estas posibilidades:

drawImage(imágen, x, y) Esta sintaxis es para dibujar una imagen en el lienzo en la posición declarada por **`x`** e **`y`**. El primer valor es una referencia a la imagen que será dibujada.

drawImage(imágen, x, y, ancho, alto) Esta sintaxis nos permite escalar la imagen antes de dibujarla en el lienzo, cambiando su tamaño con los valores de los atributos **`ancho`** y **`alto`**.

drawImage(imágen, x1, y1, ancho1, alto1, x2, y2, ancho2, alto2) Esta es la sintaxis más compleja. Hay dos valores para cada parámetro. El propósito es cortar partes de la imagen y luego dibujarlas en el lienzo con un tamaño y una posición específica. Los valores **`x1`** e **`y1`** declaran la esquina superior izquierda de la parte de la imagen que será cortada. Los valores **`ancho1`** y **`alto1`** indican el tamaño de esta pieza. El resto de los valores (**`x2`**, **`y2`**, **`ancho2`** y **`alto2`**) declaran el lugar donde la pieza será dibujada en el lienzo y su nuevo tamaño (el cual puede ser igual o diferente al original).

En cada caso, el primer atributo puede ser una referencia a una imagen en el mismo documento generada por métodos como **`getElementById()`**, o creando un nuevo objeto imagen usando métodos regulares de Javascript. No es posible usar una URL o cargar un archivo desde una fuente externa directamente con este método.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var imagen=new Image();
  imagen.src="http://www.minkbooks.com/content/snow.jpg";
  imagen.addEventListener("load", function(){
      lienzo.drawImage(imagen,20,20)
    }, false);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-22. *Trabajando con imágenes.*

Comencemos con un simple ejemplo. El código del Listado 7-22 lo único que hace es cargar la imagen y dibujarla en el lienzo. Debido a que el lienzo solo puede dibujar imágenes que ya están completamente cargadas, necesitamos controlar esta situación escuchando al evento **`load`**. Agregamos una escucha para este evento y declaramos una función anónima para responder al mismo. El método **`drawImage()`** dentro de esta función dibujará la imagen cuando fue completamente cargada.

Conceptos básicos: En el Listado 7-22, dentro del método **`addEventListener()`**, usamos una función anónima en lugar de una referencia a una función normal. En casos como éste, cuando la función es pequeña, esta técnica vuelve al código más simple y fácil de entender. Para aprender más sobre este tema, vaya a nuestro sitio web y visite los enlaces correspondientes a este capítulo.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var imagen=new Image();
  imagen.src="http://www.minkbooks.com/content/snow.jpg";
  imagen.addEventListener("load", function(){
      lienzo.drawImage(imagen,0,0,elemento.width,elemento.height)
    }, false);
}
```

```
window.addEventListener("load", iniciar, false);
```

Listado 7-23. *Ajustando la imagen al tamaño del lienzo.*

En el Listado 7-23, agregamos dos valores al método **drawImage()** utilizado previamente para cambiar el tamaño de la imagen. Las propiedades **width** y **height** retornan las medidas del lienzo, por lo que la imagen será estirada por este código hasta cubrir el lienzo por completo.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var imagen=new Image();
  imagen.src="http://www.minkbooks.com/content/snow.jpg";
  imagen.addEventListener("load", function(){
      lienzo.drawImage(imagen,135,30,50,50,0,0,200,200)
    }, false);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-24. *Extrayendo, cambiando el tamaño y dibujando.*

En el Listado 7-24 el código presenta la sintaxis más compleja del método **drawImage()**. Nueve valores fueron provistos para obtener una parte de la imagen original, cambiar su tamaño y luego dibujarla en el lienzo. Tomamos un cuadrado de la imagen original desde la posición **135,50**, con un tamaño de **50,50** pixeles. Este bloque es redimensionado a **200,200** pixeles y finalmente dibujado en el lienzo en la posición **0,0**.

Datos de imágenes

Cuando dijimos previamente que **drawImage()** era el único método disponible para dibujar imágenes en el lienzo, mentimos. Existen unos poderosos métodos para procesar imágenes en esta API que además pueden dibujarlas en el lienzo. Debido a que estos métodos no trabajan con imágenes sino con datos, nuestra declaración previa sigue siendo legítima. ¿Pero por qué desearíamos procesar datos en lugar de imágenes?

Toda imagen puede ser representada por una sucesión de números enteros representando valores rgba (cuatro valores para cada pixel). Un grupo de valores con esta información resultará en un array unidimensional que puede ser usado luego para generar una imagen. La API Canvas ofrece tres métodos para manipular datos y procesar imágenes de este modo:

getImageData(x, y, ancho, alto) Este método toma un rectángulo del lienzo del tamaño declarado por sus atributos y lo convierte en datos. Retorna un objeto que puede ser luego accedido por sus propiedades **width**, **height** y **data**.

putImageData(datosImagen, x, y) Este método convierte a los datos en **datosImagen** en una imagen y dibuja la imagen en el lienzo en la posición especificada por **x** e **y**. Este es el opuesto a **getImageData()**.

createImageData(ancho, alto) Este método crea datos para representar una imagen vacía. Todos sus pixeles serán de color negro transparente. Puede también recibir datos como atributo (en lugar de los atributos **ancho** y **alto**) y utilizar las dimensiones tomadas de los datos provistos para crear la imagen.

La posición de cada valor en el array es calculada con la fórmula **(ancho×4×y)+(x×4)**. Éste será el primer valor del pixel (rojo); para el resto tenemos que agregar 1 al resultado (por ejemplo, **(ancho×4×y)+(x×4)+1** para verde, **(ancho×4×y)+(x×4)+2** para azul, y **(ancho×4×y)+(x×4)+3** para el valor alpha (transparencia). Veamos esto en práctica:

> **IMPORTANTE:** Debido a restricciones de seguridad, no se puede extraer información del elemento **<canvas>** luego de que una imagen tomada desde una fuente externa fue dibujada en el lienzo. Solo cuando el documento y la imagen corresponden a la misma fuente (URL) el método **getImageData()** trabajará adecuadamente. Por este motivo, para probar este ejemplo tendrá que descargar la imagen desde nuestro servidor en *www.minkbooks.com/ content/snow.jpg* (o usar una imagen propia), y luego subir esta imagen, el archivo HTML y el archivo con el código Javascript a su propio servidor. Si simplemente trata de ejecutar el siguiente ejemplo en su ordenador sin seguir los pasos previos, **no funcionará**.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var imagen=new Image();
  imagen.src="snow.jpg";
  imagen.addEventListener("load", modificarimagen, false);
}
function modificarimagen(e){
  imagen=e.target;
  lienzo.drawImage(imagen,0,0);
  var info=lienzo.getImageData(0,0,175,262);

  var pos;
  for(x=0;x<=175;x++){
    for(y=0;y<=262;y++){
      pos=(info.width*4*y)+(x*4);
      info.data[pos]=255-info.data[pos];
      info.data[pos+1]=255-info.data[pos+1];
      info.data[pos+2]=255-info.data[pos+2];
    }
  }
  lienzo.putImageData(info,0,0);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-25. *Generando un negativo de la imagen.*

Esta vez tuvimos que crear una nueva función (en lugar de utilizar una función anónima) para procesar la imagen luego de que es cargada. Primero, la función **`modificarimagen()`** genera una referencia a la imagen aprovechando la propiedad **`target`** usada en capítulos previos. En el siguiente paso, usando esta referencia y el método **`drawImage()`**, la imagen es dibujada en el lienzo en la posición **`0,0`**. No hay nada inusual en esta parte del código, pero eso es algo que pronto va a cambiar.

> **IMPORTANTE:** Los archivos para este ejemplo deben ser subidos a su propio servidor para trabajar correctamente (incluyendo la imagen **`snow.jpg`** que puede descargar desde *www.minkbooks.com/content/snow.jpg*).

La imagen utilizada en nuestro ejemplo tiene un tamaño de 350 pixeles de ancho por 262 pixeles de alto, por lo que usando el método **`getImageData()`** con los valores **`0,0`** para la esquina superior izquierda y **`175,262`** para el valor horizontal y vertical, estamos extrayendo solo la mitad izquierda de la imagen original. Estos datos son grabados dentro de la variable **`info`**.

Una vez que esta información fue recolectada, es momento de manipular cada pixel para obtener el resultado que queremos (en nuestro ejemplo esto será un negativo de este trozo de la imagen).

Debido a que cada color es declarado por un valor entre 0 y 255, el valor negativo es obtenido restando el valor real a 255 con la fórmula **`color=255-color`**. Para hacerlo con cada pixel de la imagen, debemos crear dos bucles **`for`** (uno para las columnas y otro para las filas) para obtener cada color original y calcular el valor del negativo correspondiente. El bucle **`for`** para los valores **`x`** va desde 0 a 175 (el ancho de la parte de la imagen que extrajimos del lienzo) y el **`for`** para los valores **`y`** va desde 0 a 262 (el tamaño vertical de la imagen y también el tamaño vertical del trozo de imagen que estamos procesando).

Luego de que cada pixel es procesado, la variable **`info`** con los datos de la imagen es enviada al lienzo como una imagen usando el método **`putImageData()`**. La imagen es ubicada en la misma posición que la original, reemplazando la mitad izquierda de la imagen original por el negativo que acabamos de crear.

El método **`getImageData()`** retorna un objeto que puede ser procesado a través de sus propiedades (**`width`**, **`height`** y **`data`**) o puede ser usado íntegro por el método **`putImageData()`**.

Existe otra manera de extraer datos del lienzo que retorna el contenido en una cadena de texto codificada en base64. Esta cadena puede ser usada luego como fuente para otro lienzo, como fuente de un elemento HTML (por ejemplo, **`<img>`**), o incluso ser enviado al servidor o grabado en un archivo. El siguiente es el método incluido con este fin:

toDataURL(tipo) El elemento **`<canvas>`** tiene dos propiedades, **`width`** y **`height`**, y dos métodos: **`getContext()`** y **`toDataURL()`**. Este último método retorna datos en el formato data:url conteniendo una representación del contenido del lienzo en formato PNG (o el formato de imagen especificado en el atributo **`tipo`**).

Más adelante en este libro veremos algunos ejemplos de cómo usar **`toDataURL()`** y cómo puede ayudarnos a integrar esta API con otras.

Conceptos básicos: Los datos del tipo data:url son datos que son presentados en forma de cadena de texto y pueden ser incluidos en nuestros documentos como si se tratara de datos tomados de fuentes externas (por ejemplo, la fuente para imágenes insertadas con la etiqueta **`<img>`**). Para mayor información, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Patrones

Los patrones son simples adiciones que pueden mejorar nuestros trazados. Con esta herramienta podemos agregar textura a nuestras figuras utilizando una imagen. El procedimiento es similar a la creación de gradientes; los patrones son creados por el método **`createPattern()`** y luego aplicados al trazado como si fuesen un color.

createPattern(imágen, tipo) El atributo **`imágen`** es una referencia a la imagen que vamos a usar como patrón, y **`tipo`** configura el patrón por medio de cuatro valores: **`repeat`**, **`repeat-x`**, **`repeat-y`** y **`no-repeat`**.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');

  var imagen=new Image();
  imagen.src="http://www.minkbooks.com/content/bricks.jpg";
  imagen.addEventListener("load", modificarimagen, false);
}
function modificarimagen(e){
  imagen=e.target;
  var patron=lienzo.createPattern(imagen,'repeat');
  lienzo.fillStyle=patron;
  lienzo.fillRect(0,0,500,300);
}
window.addEventListener("load", iniciar, false);
```

Listado 7-26. *Agregando un patrón para nuestro trazado.*

Hágalo usted mismo: Experimente con los diferentes valores disponibles para **`createPattern()`** y también utilizando otras figuras.

7.4 Animaciones en el lienzo

Las animaciones son creadas por código Javascript convencional. No existen métodos para ayudarnos a animar figuras en el lienzo, y tampoco existe un procedimiento predeterminado para hacerlo. Básicamente, debemos borrar el área del lienzo que queremos animar, dibujar las figuras y repetir el proceso una y otra vez. Una vez que las figuras son dibujadas no se pueden mover. Solo borrando el área y dibujando las figuras nuevamente podemos construir una animación. Por esta razón, en juegos o aplicaciones que requieren grandes cantidades de objetos a ser animados, es mejor usar imágenes en lugar de figuras construidas con trazados complejos (por ejemplo, juegos normalmente utilizan imágenes PNG, que además son útiles por su capacidad de transparencia).

Existen múltiples técnicas para lograr animaciones en el mundo de la programación. Algunas son simples y otras tan complejas como las aplicaciones para las que fueron creadas. Vamos a ver un ejemplo simple utilizando el método **clearRect()** para limpiar el lienzo y dibujar nuevamente, generando una animación con solo una función, pero siempre recuerde que si su intención es crear elaborados efectos probablemente deberá adquirir un libro de programación avanzada en Javascript antes de siquiera intentarlo.

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  window.addEventListener('mousemove', animacion, false);
}
function animacion(e){
  lienzo.clearRect(0,0,300,500);

  var xraton=e.clientX;
  var yraton=e.clientY;
  var xcentro=220;
  var ycentro=150;
  var angulo=Math.atan2(xraton-xcentro,yraton-ycentro);
  var x=xcentro+Math.round(Math.sin(angulo)*10);
  var y=ycentro+Math.round(Math.cos(angulo)*10);

  lienzo.beginPath();
  lienzo.arc(xcentro,ycentro,20,0,Math.PI*2, false);
  lienzo.moveTo(xcentro+70,150);
  lienzo.arc(xcentro+50,150,20,0,Math.PI*2, false);
  lienzo.stroke();

  lienzo.beginPath();
  lienzo.moveTo(x+10,y);
  lienzo.arc(x,y,10,0,Math.PI*2, false);
  lienzo.moveTo(x+60,y);
  lienzo.arc(x+50,y,10,0,Math.PI*2, false);
  lienzo.fill();
}
window.addEventListener("load", iniciar, false);
```

Listado 7-27. *Nuestra primera animación.*

El código en el Listado 7-27 mostrará dos ojos en pantalla que miran al puntero del ratón todo el tiempo. Para mover los ojos, debemos actualizar su posición cada vez que el ratón es movido. Por este motivo agregamos una escucha para el evento **mousemove** en la función **iniciar()**. Cada vez que el puntero del ratón cambia de posición, el evento es disparado y la función **animacion()** es llamada.

La función **animacion()** comienza limpiando el lienzo con la instrucción **clearRect(0,0,300,500)**. Luego, la posición del puntero del ratón es capturada (usando las viejas propiedades **clientX** y **clientY**) y la posición del primer ojo es grabada en las variables **xcentro** e **ycentro**.

Luego de que estas variables son inicializadas, es tiempo de comenzar con las matemáticas. Usando los valores de la posición del ratón y el centro del ojo izquierdo, calculamos el ángulo de la línea invisible que va desde un punto al otro usando el método predefinido **atan2**. Este ángulo es usado en el siguiente paso para calcular el punto exacto del centro del iris del ojo izquierdo con la fórmula **xcentro + Math.round(Math.sin(angulo) × 10)**. El número **10** en la fórmula representa la distancia desde el centro del ojo al centro del iris (porque el iris no está en el centro del ojo, está siempre sobre el borde).

Con todos estos valores podemos finalmente comenzar a dibujar nuestros ojos en el lienzo. El primer trazado es para los dos círculos representando los ojos. El primer método **`arc()`** para el primer ojo es posicionado en los valores **`xcentro`** y **`ycentro`**, y el círculo para el segundo ojo es generado 50 pixeles hacia la derecha usando la instrucción **`arc(xcentro+50, 150, 20, 0, Math.PI*2, false)`**.

La parte animada del gráfico es creada a continuación con el segundo trazado. Este trazado usa las variables **`x`** e **`y`** con la posición calculada previamente a partir del ángulo. Ambos iris son dibujados como un círculo negro sólido usando **`fill()`**.

El proceso será repetido y los valores recalculados cada vez que el evento **`mousemove`** es disparado.

Hágalo usted mismo: Copie el código del Listado 7-27 en el archivo Javascript **`canvas.js`** y abra el archivo HTML con la plantilla del Listado 7-1 en su navegador.

7.5 Procesando video en el lienzo

Al igual que para animaciones, no hay ningún método especial para mostrar video en el elemento **<canvas>**. La única manera de hacerlo es tomando cada cuadro del video desde el elemento **<video>** y dibujarlo como una imagen en el lienzo usando **drawImage()**. Así que básicamente, el procesamiento de video en el lienzo es hecho con la combinación de técnicas ya estudiadas.

Construyamos una nueva plantilla y los códigos para ver de qué estamos hablando.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Video en el Lienzo</title>
  <style>
    .cajas{
      display: inline-block;
      margin: 10px;
      padding: 5px;
      border: 1px solid #999999;
    }
  </style>
  <script src="canvasvideo.js"></script>
</head>
<body>
  <section class="cajas">
    <video id="medio" width="483" height="272">
      <source src="http://www.minkbooks.com/content/trailer2.mp4">
      <source src="http://www.minkbooks.com/content/trailer2.ogg">
    </video>
  </section>
  <section class="cajas">
    <canvas id="lienzo" width="483" height="272">
      Su navegador no soporta el elemento canvas
    </canvas>
  </section>
</body>
</html>
```

Listado 7-28. *Plantilla para reproducir video en el lienzo.*

La plantilla en el Listado 7-28 incluye dos componentes específicos: el elemento **<video>** y el elemento **<canvas>**. Con la combinación de ambos vamos a procesar y mostrar video en el lienzo.

La plantilla también incluye estilos CSS embebidos para las cajas y un archivo Javascript llamado **canvasvideo.js** para el siguiente código:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  video=document.getElementById('medio');
  video.addEventListener('click', presionar, false);
}
function presionar(){
  if(!video.paused && !video.ended){
    video.pause();
    window.clearInterval(bucle);
  }else{
    video.play();
    bucle=setInterval(procesarCuadros, 33);
  }
}
function procesarCuadros(){
  lienzo.drawImage(video,0,0);
```

```
    var info=lienzo.getImageData(0,0,483,272);
    var pos;
    var gris;
    for(x=0;x<=483;x++){
      for(y=0;y<=272;y++){
        pos=(info.width*4*y)+(x*4);
        gris=parseInt(info.data[pos]*0.2989 +
                info.data[pos+1]*0.5870 + info.data[pos+2]*0.1140);
        info.data[pos]=gris;
        info.data[pos+1]=gris;
        info.data[pos+2]=gris;
      }
    }
    lienzo.putImageData(info,0,0);
  }
  window.addEventListener("load", iniciar, false);
```

Listado 7-29. *Video color convertido en blanco y negro.*

> **Hágalo usted mismo:** Cree un nuevo archivo HTML con el código del Listado 7-28 y un archivo Javascript llamado **canvasvideo.js** con el código del Listado 7-29. Para comenzar a reproducir el video, haga clic en la caja izquierda en la pantalla.

> **IMPORTANTE:** Este ejemplo usa los métodos **getImageData()** y **putImageData()** para procesar datos de imagen. Como explicamos anteriormente, estos métodos extraen información del lienzo. Debido a restricciones de seguridad, la extracción de información desde el lienzo es desactivada luego de que el elemento recibe contenido desde un origen que no es el origen del documento que lo creó (el documento pertenece a un dominio y el video a otro). Por lo tanto, para probar este ejemplo, deberá descargar el video desde nuestro sitio web (o usar el suyo propio) y luego subir cada uno de los archivos a su servidor.

Estudiemos por un momento el código del Listado 7-29. Como dijimos previamente, para procesar video en el lienzo, simplemente debemos recurrir a códigos y técnicas ya vistas. En este código estamos usando la función **presionar()** tomada del Capítulo 5 para comenzar y detener la reproducción del video haciendo clic sobre el mismo. También creamos una función llamada **procesarCuadros()** que está usando el mismo código del Listado 7-25 de este capítulo, excepto que esta vez en lugar de invertir la imagen estamos usando una fórmula para transformar todos los colores de cada cuadro del video en el correspondiente gris. Esto convertirá nuestro video color en un video blanco y negro.

La función **presionar()** cumple con dos propósitos: comenzar o detener la reproducción del video e iniciar un intervalo que ejecutará la función **procesarCuadros()** cada 33 milisegundos. Esta función toma un cuadro del elemento **<video>** y lo dibuja en el lienzo con la instrucción **drawImage(video,0,0)**. Luego los datos son extraídos del lienzo con el método **getImageData()** y cada pixel de ese cuadro es procesado por medio de dos bucles **for** (como lo hicimos en un ejemplo anterior).

El proceso utilizado para convertir cada uno de los colores que integran cada pixel en su correspondiente gris es uno de los más populares y fáciles de encontrar en Internet. La fórmula es la siguiente: **rojo × 0.2989 + verde × 0.5870 + azul × 0.1140**. Luego de que la fórmula es calculada, el resultado debe ser asignado a cada color del pixel (rojo, verde y azul), como lo hicimos en el ejemplo usando la variable **gris**.

El proceso termina cuando dibujamos nuevamente el cuadro modificado en el lienzo usando el método **putImageData()**.

> **IMPORTANTE:** Este ejemplo es con propósitos didácticos. Procesar video en tiempo real del modo en que lo hicimos no es una práctica recomendada. Dependiendo de la configuración de su ordenador y el navegador que use para correr la aplicación, probablemente note algunas demoras en el proceso. Para crear aplicaciones Javascript útiles, siempre debe considerar su rendimiento.

7.6 Referencia rápida

La API Canvas es probablemente la más compleja y extensa de todas las APIs incluidas dentro de la especificación HTML5. Provee varios métodos y propiedades para crear aplicaciones gráficas sobre el elemento **`<canvas>`**.

Métodos

Estos métodos son específicos de la API Canvas:

getContext(contexto) Este método crea el contexto para el lienzo. Puede tomar dos valores: **`2d`** y **`3d`** para gráficos en 2 y 3 dimensiones.

fillRect(x, y, ancho, alto) Este método dibujará un rectángulo sólido directamente en el lienzo en la posición indicada por **`x,y`** y el tamaño **`ancho,alto`**.

strokeRect(x, y, ancho, alto) Este método dibujará un rectángulo vacío (solo el contorno) directamente en el lienzo en la posición indicada por **`x,y`** y el tamaño **`ancho,alto`**.

clearRect(x, y, ancho, alto) Este método borra un área en el lienzo usando una figura rectangular declarada por los valores de sus atributos.

createLinearGradient(x1, y1, x2, y2) Este método crea un gradiente lineal para asignarlo a una figura como si fuese un color usando la propiedad **`fillStyle`**. Sus atributos solo especifican las posiciones de comienzo y final del gradiente (relativas al lienzo). Para declarar los colores involucrados en el gradiente, este método debe ser usado en combinación con **`addColorStop()`**.

createRadialGradient(x1, y1, r1, x2, y2, r2) Este método crea un gradiente radial para asignarlo a una figura como si fuese un color usando la propiedad **`fillStyle`**. El gradiente es construido por medio de dos círculos. Los atributos solo especifican la posición y radio de los círculos (relativos al lienzo). Para declarar los colores involucrados en el gradiente, este método debe ser usado en combinación con **`addColorStop()`**.

addColorStop(posición, color) Este método es usado para declarar los colores para el gradiente. El atributo **`posición`** es un valor entre 0.0 y 1.0, usado para determinar dónde el color comenzará la degradación.

beginPath() Este método es requerido para comenzar un nuevo trazado.

closePath() Este método puede ser usado al final de un trazado para cerrarlo. Generará una línea recta desde la última posición del lápiz hasta el punto donde el trazado comenzó. No es necesario usar este método cuando el trazado debe permanecer abierto o es dibujado en el lienzo usando **`fill()`**.

stroke() Este método es usado para dibujar un trazado como una figura vacía (solo el contorno).

fill() Este método es usado para dibujar un trazado como una figura sólida.

clip() Este método es usado para crear una máscara a partir de un trazado. Todo lo que sea enviado al lienzo luego de que este método es declarado será dibujado sólo si cae dentro de la máscara.

moveTo(x, y) Este método mueve el lápiz virtual a una nueva posición para continuar el trazado desde ese punto.

lineTo(x, y) Este método agrega líneas rectas al trazado desde la posición actual del lápiz hasta el punto indicado por los atributos **`x`** e **`y`**.

rect(x, y, ancho, alto) Este método agrega un rectángulo al trazado en la posición **`x,y`** y con un tamaño determinado por **`ancho,alto`**.

arc(x, y, radio, ángulo inicio, ángulo final, dirección) Este método agrega un arco al trazado. El centro del arco es determinado por **`x`** e **`y`**, los ángulos son definidos en radianes, y la **`dirección`** es un valor booleano para determinar si el arco será dibujado en el mismo sentido o el opuesto a las agujas del reloj. Para convertir grados en radianes, use la fórmula: **`Math.PI/180×grados`**.

quadraticCurveTo(cpx, cpy, x, y) Este método agrega una curva Bézier cuadrática al trazado. Comienza desde la posición actual del lápiz y termina en el punto **`x,y`**. Los atributos **`cpx`** y **`cpy`** especifican la posición del punto de control que dará forma a la curva.

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) Este método agrega una curva Bézier cúbica al trazado. Comienza desde la posición actual del lápiz y termina en el punto **`x,y`**. Los atributos **`cp1x`**, **`cp1y`**, **`cp2x`**, y **`cp2y`** especifican la posición de los dos puntos de control que darán forma a la curva.

strokeText(texto, x, y, máximo) Este método dibuja un texto vacío (solo el contorno) directamente en el lienzo. El atributo **`máximo`** es opcional y determina el máximo tamaño del texto en pixeles.

fillText(texto, x, y, máximo) Este método dibuja un texto sólido directamente en el lienzo. El atributo **máximo** es opcional y determina el máximo tamaño del texto en pixeles.

measureText(texto) Este método calcula el tamaño del área que un texto ocupará en el lienzo usando los estilos vigentes. La propiedad **width** es usada para retornar el valor.

translate(x, y) Este método mueve el origen del lienzo al punto **x,y**. La posición inicial del origen (0,0) es la esquina superior izquierda del área generada por el elemento **<canvas>**.

rotate(angle) Este método es usado para rotar el lienzo alrededor del origen. El ángulo debe ser declarado en radianes. Para convertir grados en radianes, use la fórmula: **Math.PI/180×grados**.

scale(x, y) Este método cambia la escala del lienzo. Los valores por defecto son (1.0, 1.0). Los valores provistos pueden ser negativos.

transform(m1, m2, m3, m4, dx, dy) Este método modifica la matriz de transformación del lienzo. La nueva matriz es calculada sobre la anterior.

setTransform(m1, m2, m3, m4, dx, dy) Este método modifica la matriz de transformación del lienzo. Reinicia los valores anteriores y declara los nuevos.

save() Este método graba el estado del lienzo, incluyendo la matriz de transformación, propiedades de estilo y la máscara.

restore() Este método restaura el último estado del lienzo grabado, incluyendo la matriz de transformación, propiedades de estilo y la máscara.

drawImage() Esta método dibujará una imagen en el lienzo. Existen tres sintaxis posibles. La sintaxis **drawImage(imagen,x,y)** dibuja la imagen en la posición **x,y**. La sintaxis **drawImage(imagen,x,y,ancho,alto)** dibuja la imagen en la posición **x,y** con un nuevo tamaño declarado por **ancho,alto**. Y la sintaxis **drawImage(imagen, x1, y1, ancho1, alto1, x2, y2, ancho2, alto2)** toma una porción de la imagen original determinada por **x1,y1,ancho1,alto1** y la dibuja en el lienzo en la posición **x2,y2** y el nuevo tamaño **ancho2,alto2**.

getImageData(x, y, ancho, alto) Este método toma una porción del lienzo y la graba como datos en un objeto. Los valores del objeto son accesibles a través de las propiedades **width**, **height** y **data**. Las primeras dos propiedades retornan el tamaño de la porción de la imagen tomada, y **data** retorna la información como un array con valores representando los colores de cada pixel. Este valor puede ser accedido usando la fórmula **(ancho×4×y)+(x×4)**.

putImageData(datosImagen, x, y) Este método dibuja en el lienzo la imagen representada por la información en **datosImagen**.

createImageData(ancho, alto) Este método crea una nueva imagen en formato de datos. Todos los pixeles son inicializados en color negro transparente. Puede tomar datos de imagen como atributo en lugar de **ancho** y **alto**. En este caso la nueva imagen tendrá el tamaño determinado por los datos provistos.

createPattern(imagen, tipo) Este método crea un patrón desde una imagen que luego podrá ser asignado a una figura usando la propiedad **fillStyle**. Los valores posibles para el atributo **tipo** son **repeat**, **repeat-x**, **repeat-y** y **no-repeat**.

Propiedades

La siguiente lista de propiedades es específica para la API Canvas:

strokeStyle Esta propiedad declara el color para las líneas de las figuras. Puede recibir cualquier valor CSS, incluidas funciones como **rgb()** y **rgba()**.

fillStyle Esta propiedad declara el color para el interior de figuras sólidas. Puede recibir cualquier valor CSS, incluidas funciones como **rgb()** y **rgba()**. Es también usada para asignar gradientes y patrones a figuras (estos estilos son primero asignados a una variable y luego esa variable es declarada como el valor de esta propiedad).

globalAlpha Esta propiedad es usada para determinar el nivel de transparencia de las figuras. Recibe valores entre 0.0 (completamente opaco) y 1.0 (completamente transparente).

lineWidth Esta propiedad especifica el grosor de la línea. Por defecto el valor es 1.0.

lineCap - Esta propiedad determina la forma de la terminación de las líneas. Se pueden utilizar tres valores: **butt** (terminación normal), **round** (termina la línea con un semicírculo) y **square** (termina la línea con un cuadrado).

lineJoin Esta propiedad determina la forma de la conexión entre líneas. Se pueden utilizar tres valores: **`round`** (la unión es redondeada), **`bevel`** (la unión es cortada) y **`miter`** (la unión es extendida hasta que ambas líneas alcanzan un punto en común).

miterLimit Esta propiedad determina cuánto se extenderán las líneas cuando la propiedad **`lineJoin`** es declarada como **`miter`**.

font Esta propiedad es similar a la propiedad **`font`** de CSS y utiliza la misma sintaxis para declarar los estilos del texto.

textAlign Esta propiedad determina cómo el texto será alineado. Los posibles valores son **`start`**, **`end`**, **`left`**, **`right`** y **`center`**.

textBaseline Esta propiedad determina el alineamiento vertical para el texto. Los posibles valores son: **`top`**, **`hanging`**, **`middle`**, **`alphabetic`**, **`ideographic`** y **`bottom`**.

shadowColor Esta propiedad establece el color para la sombra. Utiliza valores CSS.

shadowOffsetX Esta propiedad declara la distancia horizontal entre la sombra y el objeto.

shadowOffsetY Esta propiedad declara la distancia vertical entre la sombra y el objeto.

shadowBlur Esta propiedad recibe un valor numérico para generar un efecto de difuminación para la sombra.

globalCompositeOperation Esta propiedad determina cómo las nuevas figuras serán dibujadas en el lienzo considerando las figuras ya existentes. Puede recibir varios valores: **`source-over`**, **`source-in`**, **`source-out`**, **`source-atop`**, **`lighter`**, **`xor`**, **`destination-over`**, **`destination-in`**, **`destination-out`**, **`destination-atop`**, **`darker`** y **`copy`**. El valor por defecto es **`source-over`**, lo que significa que las nuevas formas son dibujadas sobre las anteriores.

Capítulo 8
API Drag and Drop

8.1 Arrastrar y soltar en la web

Arrastrar un elemento desde un lugar y luego soltarlo en otro es algo que hacemos todo el tiempo en aplicaciones de escritorio, pero ni siquiera imaginamos hacerlo en la web. Esto no es debido a que las aplicaciones web son diferentes sino porque desarrolladores nunca contaron con una tecnología estándar disponible para ofrecer esta herramienta.

Ahora, gracias a la API Drag and Drop, introducida por la especificación HTML5, finalmente tenemos la oportunidad de crear software para la web que se comportará exactamente como las aplicaciones de escritorio que usamos desde siempre.

Nuevos eventos

Uno de los más importantes aspectos de esta API es un conjunto de siete nuevos eventos introducidos para informar sobre cada una de las situaciones involucradas en el proceso. Algunos de estos eventos son disparados por la fuente (el elemento que es arrastrado) y otros son disparados por el destino (el elemento en el cual el elemento arrastrado será soltado). Por ejemplo, cuando el usuario realiza una operación de arrastrar y soltar, el elemento origen (el que es arrastrado) dispara estos tres eventos:

dragstart Este evento es disparado en el momento en el que el arrastre comienza. Los datos asociados con el elemento origen son definidos en este momento en el sistema.

drag Este evento es similar al evento **`mousemove`**, excepto que será disparado durante una operación de arrastre por el elemento origen.

dragend Cuando la operación de arrastrar y soltar finaliza (sea la operación exitosa o no) este evento es disparado por el elemento origen.

Y estos son los eventos disparados por el elemento destino (donde el origen será soltado) durante la operación:

dragenter Cuando el puntero del ratón entra dentro del área ocupada por los posibles elementos destino durante una operación de arrastrar y soltar, este evento es disparado.

dragover Este evento es similar al evento **`mousemove`**, excepto que es disparado durante una operación de arrastre por posibles elementos destino.

drop Cuando el elemento origen es soltado durante una operación de arrastrar y soltar, este evento es disparado por el elemento destino.

dragleave Este evento es disparado cuando el ratón sale del área ocupada por un elemento durante una operación de arrastrar y soltar. Este evento es generalmente usado junto con **`dragenter`** para mostrar una ayuda visual al usuario que le permita identificar el elemento destino (donde soltar).

Antes de trabajar con esta nueva herramienta, existe un aspecto importante que debemos considerar. Los navegadores realizan acciones por defecto durante una operación de arrastrar y soltar.

Para obtener el resultado que queremos, necesitamos prevenir en algunas ocasiones este comportamiento por defecto y personalizar las reacciones del navegador. Para algunos eventos, como **`dragenter`**, **`dragover`** y **`drop`**, la prevención es necesaria, incluso cuando una acción personalizada ya fue especificada.

Veamos cómo debemos proceder usando un ejemplo simple.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="cajasoltar">
    Arrastre y suelte la imagen aquí
```

```
    </section>
    <section id="cajaimagenes">
      <img id="imagen" src="http://www.minkbooks.com/content/
                                                    monster1.gif">
    </section>
  </body>
  </html>
```

Listado 8-1. *Plantilla para la operación arrastrar y soltar.*

El documento HTML del Listado 8-1 incluye un elemento **<section>** identificado como **cajasoltar** y una imagen. El elemento **<section>** será usado como elemento destino y la imagen será el elemento a arrastrar. También incluimos dos archivos para estilos CSS y el código javascript que se hará cargo de la operación.

```
#cajasoltar{
  float: left;
  width: 500px;
  height: 300px;
  margin: 10px;
  border: 1px solid #999999;
}
#cajaimagenes{
  float: left;
  width: 320px;
  margin: 10px;
  border: 1px solid #999999;
}
#cajaimagenes > img{
  float: left;
  padding: 5px;
}
```

Listado 8-2. *Estilos para la plantilla (dragdrop.css).*

Las reglas en el Listado 8-2 simplemente otorgan estilos a las cajas que nos servirán para identificar el elemento a arrastrar y el destino.

```
function iniciar(){
  origen1=document.getElementById('imagen');
  origen1.addEventListener('dragstart', arrastrado, false);

  destino=document.getElementById('cajasoltar');
  destino.addEventListener('dragenter', function(e){
                              e.preventDefault(); }, false);
  destino.addEventListener('dragover', function(e){
                              e.preventDefault(); }, false);
  destino.addEventListener('drop', soltado, false);
}
function arrastrado(e){
  var codigo='<img src="'+origen1.getAttribute('src')+'">';
  e.dataTransfer.setData('Text', codigo);
}
function soltado(e){
  e.preventDefault();
  destino.innerHTML=e.dataTransfer.getData('Text');
}
window.addEventListener('load', iniciar, false);
```

Listado 8-3. *Código elemental para una operación arrastrar y soltar.*

Existen algunos atributos que podemos usar en los elementos HTML para configurar el proceso de una operación arrastrar y soltar, pero básicamente todo puede ser hecho desde código Javascript. En el Listado 8-3, presentamos tres funciones: la función **iniciar()** agrega las escuchas para los eventos necesarios en esta operación, y las funciones

arrastrado() y **soltado()** generan y reciben la información que es transmitida por este proceso.

Para que una operación arrastrar y soltar se realice normalmente, debemos preparar la información que será compartida entre el elemento origen y el elemento destino. Para lograr esto, una escucha para el evento **dragstart** fue agregada. La escucha llama a la función **arrastrado()** cuando el evento es disparado y la información a ser compartida es preparada en esta función usando **setData()**.

La operación soltar no es normalmente permitida en la mayoría de los elementos de un documento por defecto. Por este motivo, para hacer esta operación disponible en nuestro elemento destino, debemos prevenir el comportamiento por defecto del navegador. Esto fue hecho agregando una escucha para los eventos **dragenter** y **dragover** y ejecutando el método **preventDefault()** cuando son disparados.

Finalmente, una escucha para el evento **drop** fue agregada para llamar a la función **soltado()** que recibirá y procesará los datos enviados por el elemento origen.

> **Conceptos básicos:** Para responder a los eventos **dragenter** y **dragover** usamos una función anónima y llamamos en su interior al método **preventDefault()** que cancela el comportamiento por defecto del navegador. La variable **e** fue enviada para referenciar al evento dentro de la función. Para obtener más información acerca de funciones anónimas, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Cuando el elemento origen comienza a ser arrastrado, el evento **dragstart** es disparado y la función **arrastrado()** es llamada. En esta función obtenemos el valor del atributo **src** del elemento que está siendo arrastrado y declaramos los datos que serán transferidos usando el método **setData()** del objeto **dataTransfer**. Desde el otro lado, cuando un elemento es soltado dentro del elemento destino, el evento **drop** es disparado y la función **soltado()** es llamada. Esta función modifica el contenido del elemento destino con la información obtenida por el método **getData()**. Los navegadores también realizan acciones por defecto cuando estos eventos son disparados (por ejemplo, abrir un enlace o actualizar la ventana para mostrar la imagen que fue soltada) por lo que debemos prevenir este comportamiento usando el método **preventDefault()**, como ya hicimos para otros eventos anteriormente.

> **Hágalo usted Mismo:** Cree un archivo HTML con la plantilla del Listado 8-1, un archivo CSS llamado **dragdrop.css** con los estilos del Listado 8-2, y un archivo Javascript llamado **dragdrop.js** con el código del Listado 8-3. Para probar el ejemplo, abra el archivo HTML en su navegador y arrastre la imagen hacia el cuadro de la izquierda.

dataTransfer

Este es el objeto que contendrá la información en una operación arrastrar y soltar. El objeto **dataTransfer** tiene varios métodos y propiedades asociados. Ya utilizamos los métodos **setData()** y **getData()** en nuestro ejemplo del Listado 8-3. Junto con **clearData()**, estos son los métodos a cargo de la información que es transferida:

setData(tipo, dato) Este método es usado para declarar los datos a ser enviados y su tipo. El método puede recibir tipos de datos regulares (como **text/plain**, **text/html** o **text/uri-list**), tipos de datos especiales (como **URL** o **Text**) o incluso tipos de datos personalizados. Un método **setData()** debe ser llamado por cada tipo de datos que queremos enviar en la misma operación.

getData(tipo) Este método retorna los datos enviados por el origen, pero solo del tipo especificado.

clearData() Este método remueve los datos del tipo especificado.

En la función **arrastrado()** del Listado 8-3, creamos un pequeño código HTML que incluye el valor del atributo **src** del elemento que comenzó a ser arrastrado, grabamos este código en la variable **codigo** y luego enviamos esta variable como el dato a ser transferido usando el método **setData()**. Debido a que estamos enviando texto, declaramos el tipo de dato como **Text**.

> **IMPORTANTE:** Podríamos haber usado un tipo de datos más apropiado en nuestro ejemplo, como **text/html** o incluso un tipo personalizado, pero varios navegadores solo admiten un número limitado de tipos en este momento, por lo que el tipo **Text** hace a nuestra pequeña aplicación más compatible y la deja lista para ser ejecutada.

Cuando recuperamos los datos en la función **soltado()** usando el método **getData()**, tenemos que especificar el tipo de datos a ser leído. Esto es debido a que diferentes clases de datos pueden ser enviados por el mismo elemento. Por ejemplo, una imagen podría enviar la imagen misma, la URL y un texto describiendo la imagen. Toda esta información puede ser enviada usando varias declaraciones de **setData()** con diferentes tipos de valores y luego recuperada por **getData()** especificando los mismo tipos.

> **IMPORTANTE:** Para obtener mayor información acerca de tipos de datos para la operación arrastrar y soltar, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

El objeto **dataTransfer** tiene algunos métodos y propiedades más que a veces podrían resultar útil para nuestras aplicaciones:

setDragImage(elemento, x, y) Algunos navegadores muestran una imagen en miniatura junto al puntero del ratón que representa al elemento que está siendo arrastrado. Este método es usado para personalizar esa imagen y seleccionar la posición la posición en la que será mostrada relativa al puntero del ratón. Esta posición es determinada por los atributos **x** e **y**.

types Esta propiedad retorna un array conteniendo los tipos de datos que fueron declarados durante el evento **dragstart** (por el código o el navegador). Podemos grabar este array en una variable (**lista=dataTransfer.types**) y luego leerlo con un bucle **for**.

files Esta propiedad retorna un array conteniendo información acerca de los archivos que están siendo arrastrados.

dropEffect Esta propiedad retorna el tipo de operación actualmente seleccionada. Los posibles valores son **none**, **copy**, **link** y **move**.

effectAllowed Esta propiedad retorna los tipos de operaciones que están permitidas. Puede ser usada para cambiar las operaciones permitidas. Los posibles valores son: **none**, **copy**, **copyLink**, **copyMove**, **link**, **linkMove**, **move**, **all** y **uninitialized**.

Aplicaremos algunos de estos métodos y propiedades en los siguientes ejemplos.

dragenter, dragleave y dragend

Nada fue hecho aún con el evento **dragenter**. Solo cancelamos el comportamiento por defecto de los navegadores cuando este evento es disparado para prevenir efectos no deseados. Y tampoco aprovechamos los eventos **dragleave** y **dragend**. Estos son eventos importantes que nos permitirán ayudar al usuario cuando se encuentra arrastrando objetos por la pantalla.

```
function iniciar(){
  origen1=document.getElementById('imagen');
  origen1.addEventListener('dragstart', arrastrado, false);
  origen1.addEventListener('dragend', finalizado, false);

  soltar=document.getElementById('cajasoltar');
  soltar.addEventListener('dragenter', entrando, false);
  soltar.addEventListener('dragleave', saliendo, false);
    soltar.addEventListener('dragover', function(e){
                                e.preventDefault(); }, false);
  soltar.addEventListener('drop', soltado, false);
}
function entrando(e){
  e.preventDefault();
  soltar.style.background='rgba(0,150,0,.2)';
}

function saliendo(e){
  e.preventDefault();
  soltar.style.background='#FFFFFF';
}
function finalizado(e){
  elemento=e.target;
  elemento.style.visibility='hidden';
}
function arrastrado(e){
  var codigo='<img src="'+origen1.getAttribute('src')+'">';
  e.dataTransfer.setData('Text', codigo);
}
function soltado(e){
  e.preventDefault();
  soltar.style.background='#FFFFFF';
  soltar.innerHTML=e.dataTransfer.getData('Text');
}
```

```
window.addEventListener('load', iniciar, false);
```

Listado 8-4. *Controlando todo el proceso de arrastrar y soltar.*

El código Javascript del Listado 8-4 reemplaza al código del Listado 8-3. En este nuevo ejemplo, agregamos dos funciones para el elemento destino y una para el elemento origen. Las funciones **`entrando()`** y **`saliendo()`** cambiarán el color de fondo del elemento destino cada vez que el puntero del ratón esté arrastrando un objeto y entre o salga del área ocupada por este elemento (estas acciones disparan los eventos **`dragenter`** y **`dragleave`**). Además, la función **`finalizado()`** será llamada por la escucha del evento **`dragend`** cuando el objeto arrastrado es soltado. Note que este evento o la función misma no controlan si el proceso fue exitoso o no. Este control lo deberemos hacer nosotros en el código.

Gracias a los eventos y funciones agregadas, cada vez que el ratón arrastra un objeto y entra en el área del elemento destino, este elemento se volverá verde, y cuando el objeto es soltado la imagen original es borrada de la pantalla. Estos cambios visibles no están afectando el proceso de arrastrar y soltar, pero sí están ofreciendo una guía clara para el usuario durante la operación.

Para prevenir acciones por defecto del navegador, tenemos que usar el método **`preventDefault()`** en cada función, incluso cuando acciones personalizadas fueron declaradas.

> **Hágalo usted mismo:** Copie el código del Listado 8-4 dentro del archivo Javascript, abra el documento HTML del Listado 8-1 en su navegador, y arrastre la imagen que aparece en la pantalla dentro de la caja ubicada a su izquierda.

Seleccionando un origen válido

No existe ningún método específico para detectar si el elemento origen es válido o no. No podemos confiar en la información retornada por el método **`getData()`** porque incluso cuando podemos recuperar solo los datos del tipo especificado, otras fuentes podrían originar el mismo tipo y proveer datos que no esperábamos. Hay una propiedad del objeto **`dataTransfer`** llamada **`types`** que retorna un array con la lista de tipos configurados durante el evento **`dragstart`**, pero también es inútil para propósitos de validación.

Por esta razón, las técnicas para seleccionar y validar los datos transferidos en una operación arrastrar y soltar son variados, y pueden ser tan simples o complejos como necesitemos.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="cajasoltar">
    Arrastre y suelte las imágenes aquí
  </section>
  <section id="cajaimagenes">
    <img id="imagen1"
              src="http://www.minkbooks.com/content/monster1.gif">
    <img id="imagen2"
            src="http://www.minkbooks.com/content/monster2.gif">
    <img id="imagen3"
            src="http://www.minkbooks.com/content/monster3.gif">
    <img id="imagen4"
            src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Listado 8-5. *Nueva plantilla con varias imágenes para arrastrar.*

Usando la nueva plantilla HTML del Listado 8-5 vamos a filtrar los elementos a ser soltados dentro del elemento destino controlando el atributo **`id`** de la imagen. El siguiente código Javascript indicará cuál imagen puede ser soltada y cuál no:

```
function iniciar(){
  var imagenes=document.querySelectorAll('#cajaimagenes > img');
  for(var i=0; i<imagenes.length; i++){
    imagenes[i].addEventListener('dragstart', arrastrado, false);
  }

  soltar=document.getElementById('cajasoltar');
  soltar.addEventListener('dragenter', function(e){
                                  e.preventDefault(); }, false);
  soltar.addEventListener('dragover', function(e){
                                  e.preventDefault(); }, false);
  soltar.addEventListener('drop', soltado, false);
}
function arrastrado(e){
  elemento=e.target;
  e.dataTransfer.setData('Text', elemento.getAttribute('id'));
}
function soltado(e){
  e.preventDefault();
  var id=e.dataTransfer.getData('Text');
  if(id!="imagen4"){
    var src=document.getElementById(id).src;
    soltar.innerHTML='<img src="'+src+'">';
  }else{
    soltar.innerHTML='la imagen no es admitida';
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 8-6. *Enviando el valor del atributo `id`.*

No han cambiado muchas cosas en el Listado 8-6 de anteriores listados. En este código estamos usando el método **`querySelectorAll()`** para agregar una escucha para el evento **`dragstart`** a cada imagen dentro del elemento **`cajaimagenes`**, enviando el valor del atributo **`id`** con **`setData()`** cada vez que una imagen es arrastrada, y controlando el valor de **`id`** en la función **`soltado()`** para evitar que el usuario arrastre y suelte la imagen con el atributo igual a **`"imagen4"`** (el mensaje `"la imagen no es admitida"` es mostrado dentro del elemento destino cuando el usuario intenta arrastrar y soltar esta imagen en particular).

Este es, por supuesto, un filtro extremadamente sencillo. Puede usar el método **`querySelectorAll()`** en la función **`soltado()`** para controlar que la imagen recibida es una de las que se encuentran dentro del elemento **`cajaimagenes`**, por ejemplo, o usar propiedades del objeto **`dataTransfer`** (como **`types`** o **`files`**), pero es siempre un proceso personalizado. En otras palabras, deberemos hacernos cargo nosotros mismos de realizar este control.

setDragImage()

Cambiar la imagen en miniatura que es mostrada junto al puntero del ratón en una operación arrastrar y soltar puede parecer inútil, pero en ocasiones nos evitará dolores de cabeza. El método **`setDragImage()`** no solo nos permite cambiar la imagen sino también recibe dos atributos, **`x`** e **`y`**, para especificar la posición de esta imagen relativa al puntero. Algunos navegadores generan una imagen en miniatura por defecto a partir del objeto original que es arrastrado, pero su posición relativa al puntero del ratón es determinada por la posición del puntero cuando el proceso comienza. El método **`setDragImage()`** nos permite declarar una posición específica que será la misma para cada operación arrastrar y soltar.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="cajasoltar">
```

```
    <canvas id="lienzo" width="500" height="300"></canvas>
  </section>
  <section id="cajaimagenes">
    <img id="imagen1"
           src="http://www.minkbooks.com/content/monster1.gif">
    <img id="imagen2"
           src="http://www.minkbooks.com/content/monster2.gif">
    <img id="imagen3"
           src="http://www.minkbooks.com/content/monster3.gif">
    <img id="imagen4"
           src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Listado 8-7. *<canvas> como elemento destino.*

Con el nuevo documento HTML del Listado 8-7 vamos a estudiar la importancia del método **setDragImage()** usando un elemento **<canvas>** como el elemento destino.

```
function iniciar(){
  var imagenes=document.querySelectorAll('#cajaimagenes > img');
  for(var i=0; i<imagenes.length; i++){
    imagenes[i].addEventListener('dragstart', arrastrado, false);
    imagenes[i].addEventListener('dragend', finalizado, false);
  }
  soltar=document.getElementById('lienzo');
  lienzo=soltar.getContext('2d');

  soltar.addEventListener('dragenter', function(e){
                               e.preventDefault(); }, false);
  soltar.addEventListener('dragover', function(e){
                               e.preventDefault(); }, false);
  soltar.addEventListener('drop', soltado, false);
}
function finalizado(e){
  elemento=e.target;
  elemento.style.visibility='hidden';
}
function arrastrado(e){
  elemento=e.target;
  e.dataTransfer.setData('Text', elemento.getAttribute('id'));
  e.dataTransfer.setDragImage(e.target, 0, 0);
}
function soltado(e){
  e.preventDefault();
  var id=e.dataTransfer.getData('Text');
  var elemento=document.getElementById(id);

  var posx=e.pageX-soltar.offsetLeft;
  var posy=e.pageY-soltar.offsetTop;

  lienzo.drawImage(elemento,posx,posy);
}
window.addEventListener('load', iniciar, false);
```

Listado 8-8. *Una pequeña aplicación para arrastrar y soltar.*

Probablemente, con este ejemplo, nos estemos acercando a lo que sería una aplicación de la vida real. El código del Listado 8-8 controlará tres diferentes aspectos del proceso. Cuando la imagen es arrastrada, la función **arrastrado()** es llamada y en su interior una imagen miniatura es generada con el método **setDragImage()**. El código también crea el contexto para trabajar con el lienzo y dibuja la imagen soltada usando el método **drawImage()** estudiado en el capítulo anterior. Al final de todo el proceso la imagen original es ocultada usando la función **finalizado()**.

Para la imagen miniatura personalizada usamos el mismo elemento que está siendo arrastrado, pero declaramos la posición relativa al puntero del ratón como **0,0**. Gracias a esto ahora sabremos siempre cual es exactamente la ubicación de la imagen miniatura. Aprovechamos este dato importante dentro de la función **soltado()**. Usando la misma técnica introducida en el capítulo anterior, calculamos dónde el objeto es soltado dentro del lienzo y dibujamos la imagen en ese lugar preciso. Si prueba este ejemplo en navegadores que ya aceptan el método **setDragImage()** (por ejemplo, Firefox 4+), verá que la imagen es dibujada en el lienzo exactamente en la posición de la imagen miniatura que acompaña al puntero del ratón, haciendo fácil para el usuario seleccionar el lugar adecuado donde soltarla.

> **IMPORTANTE:** El código en el Listado 8-8 usa el evento **dragend** para ocultar la imagen original cuando la operación termina. Este evento es disparado por el elemento origen cuando una operación de arrastre finaliza, incluso cuando no fue exitosa. En nuestro ejemplo la imagen será ocultada en ambos casos, éxito o fracaso. Usted deberá crear los controles adecuados para actuar solo en caso de éxito.

Archivos

Posiblemente la característica más interesante de la API Drag and Drop es la habilidad de trabajar con archivos. La API no está solo disponible dentro del documento, sino también integrada con el sistema, permitiendo a los usuarios arrastrar elementos desde el navegador hacia otras aplicaciones y viceversa. Y normalmente los elementos más requeridos desde aplicaciones externas son archivos.

Como vimos anteriormente, existe una propiedad especial en el objeto **dataTransfer** que retornará un array conteniendo la lista de archivos que están siendo arrastrados. Podemos usar esta información para construir complejos códigos que trabajan con archivos o subirlos a un servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="cajasoltar">
    Arrastre y suelte archivos en este espacio
  </section>
</body>
</html>
```

Listado 8-9. *Plantilla simple para arrastrar archivos.*

El documento HTML del Listado 8-9 genera simplemente una caja para soltar los archivos arrastrados. Los archivos serán arrastrados desde una aplicación externa (por ejemplo, el Explorador de Archivos de Windows). Los datos provenientes de los archivos serán procesados por el siguiente código:

```
function iniciar(){
  soltar=document.getElementById('cajasoltar');
  soltar.addEventListener('dragenter', function(e){
                                e.preventDefault(); }, false);
  soltar.addEventListener('dragover', function(e){
                                e.preventDefault(); }, false);
  soltar.addEventListener('drop', soltado, false);
}
function soltado(e){
  e.preventDefault();
  var archivos=e.dataTransfer.files;
  var lista='';
  for(var f=0;f<archivos.length;f++){
    lista+='Archivo: '+archivos[f].name+' '+archivos[f].size+'<br>';
  }
  soltar.innerHTML=lista;
}
```

```
window.addEventListener('load', iniciar, false);
```

Listado 8-10. *Procesando los datos en la propiedad* `files`.

La información retornada por la propiedad **`files`** del objeto **`dataTransfer`** puede ser grabada en una variable y luego leída por un bucle **`for`**. En el código del Listado 8-10, solo mostramos el nombre y el tamaño del archivo en el elemento destino usando las propiedades **`name`** y **`size`**. Para aprovechar esta información y construir aplicaciones más complejas, necesitaremos recurrir a otras APIs y técnicas de programación, como veremos más adelante en este libro.

> **Hágalo usted mismo:** Cree nuevos archivos con los códigos de los Listados 8-9 y 8-10, y abra la plantilla en su navegador. Arrastre archivos desde el Explorador de Archivos o cualquier otra aplicación similar dentro del elemento destino. Luego de esta acción debería ver en pantalla una lista con el nombre y tamaño de cada archivo arrastrado.

8.2 Referencia rápida

La API Drag and Drop introduce eventos específicos, métodos y propiedades para construir aplicaciones que incorporan la capacidad de arrastrar y soltar elementos en pantalla.

Eventos

Existen siete eventos para esta API:

dragstart Este evento es disparado por el elemento origen cuando la operación de arrastre comienza.

drag Este evento es disparado por el elemento origen mientras una operación de arrastre se está realizando.

dragend Este evento es disparado por el elemento origen cuando una operación de arrastre es terminada, ya sea porque la acción de soltar fue exitosa o porque la operación de arrastre fue cancelada.

dragenter Este evento es disparado por el elemento destino cuando el puntero del ratón entra en el área ocupada por este elemento. Este evento siempre tiene que ser cancelado usando el método **`preventDefault()`**.

dragover Este evento es disparado periódicamente por el elemento destino cuando el puntero del ratón está sobre él. Este evento siempre tiene que ser cancelado usando el método **`preventDefault()`**.

drop Este evento es disparado por el elemento destino cuando el elemento origen es soltado en su interior. Este evento siempre tiene que ser cancelado usando el método **`preventDefault()`**.

dragleave Este evento es disparado por el elemento destino cuando el puntero del ratón sale del área ocupada por el mismo.

Métodos

La siguiente es una lista de los métodos más importantes incorporados por esta API:

setData(tipo, dato) Este método es usado para preparar los datos a ser enviados cuando el evento **`dragstart`** es disparado. El atributo **`tipo`** puede ser cualquier tipo de datos regular (como **`text/plain`** o **`text/html`**) o un tipo de datos personalizado.

getData(tipo) Este método retorna los datos del tipo especificado. Es usado cuando un evento **`drop`** es disparado.

clearData(type) Este método remueve los datos del tipo especificado.

setDragImage(elemento, x, y) Este método reemplaza la imagen en miniatura creada por el navegador en la operación arrastrar y soltar por una imagen personalizada. También declara la posición que esta imagen tendrá con respecto al puntero del ratón.

Propiedades

El objeto **`dataTransfer`**, que contiene los datos transferidos en una operación arrastrar y soltar, también introduce algunas propiedades útiles:

types Esta propiedad retorna un array con todos los tipos establecidos durante el evento **`dragstart`**.

files Esta propiedad retorna un array con información acerca de los archivos que están siendo arrastrados.

dropEffect Esta propiedad retorna el tipo de operación actualmente seleccionada. Los valores posibles son: **`none`**, **`copy`**, **`link`** y **`move`**.

effectAllowed Esta propiedad retorna los tipos de operación que están permitidos. Puede ser declarada para cambiar las operaciones permitidas. Los posibles valores son: **`none`**, **`copy`**, **`copyLink`**, **`copyMove`**, **`link`**, **`linkMove`**, **`move`**, **`all`** y **`uninitialized`**.

Capítulo 9
API Geolocation

9.1 Encontrando su lugar

La API Geolocation fue diseñada para que los navegadores puedan proveer un mecanismo de detección por defecto que permita a los desarrolladores determinar la ubicación física real del usuario. Previamente solo contábamos con la opción de construir una gran base de datos con información sobre direcciones IP y programar códigos exigentes dentro del servidor que nos darían una idea aproximada de la ubicación del usuario (generalmente tan imprecisa como su país).

Esta API aprovecha nuevos sistemas, como triangulación de red y GPS, para retornar una ubicación precisa del dispositivo que está accediendo a la aplicación. La valiosa información retornada nos permite construir aplicaciones que se adaptarán a las particulares necesidades del usuario o proveerán información localizada de forma automática.

Tres métodos específicos son provistos para usar la API:

getCurrentPosition(ubicación, error, configuración) Este es el método utilizado para consultas individuales. Puede recibir hasta tres atributos: una función para procesar la ubicación retornada, una función para procesar los errores retornados, y un objeto para configurar cómo la información será adquirida. Solo el primer atributo es obligatorio para que el método trabaje correctamente.

watchPosition(ubicación, error, configuración) Este método es similar al anterior, excepto que comenzará un proceso de vigilancia para la detección de nuevas ubicaciones. Trabaja de forma similar que el conocido método **`setInterval()`** de Javascript, repitiendo el proceso automáticamente en determinados períodos de tiempo de acuerdo a la configuración por defecto o a los valores de sus atributos.

clearWatch(id) El método **`watchPosition()`** retorna un valor que puede ser almacenado en una variable para luego ser usado como referencia pro el método **`clearWatch()`** y así detener la vigilancia. Este método es similar a **`clearInterval()`** usado para detener los procesos comenzados por **`setInterval()`**.

getCurrentPosition(ubicación)

Como dijimos, solo el primer atributo es requerido para que trabaje correctamente el método **`getCurrentPosition()`**. Este atributo es una función que recibirá un objeto llamado **`Position`**, el cual contiene toda la información retornada por los sistemas de ubicación.

El objeto **`Position`** tiene dos atributos:

coords Este atributo contiene un grupo de valores que establecen la ubicación del dispositivo y otros datos importantes. Los valores son accesibles a través de siete atributos internos: **`latitude`** (latitud), **`longitude`** (longitud), **`altitude`** (altitud en metros), **`accuracy`** (exactitud en metros), **`altitudeAccuracy`** (exactitud de la altitud en metros), **`heading`** (dirección en grados) y **`speed`** (velocidad en metros por segundo).

timestamp Este atributo indica el momento en el que la información fue adquirida (en formato timestamp).

Este objeto, como dijimos, es pasado a la función que definimos como atributo del método **`getCurrentPosition()`** y luego sus datos son accedidos y procesados en esta función. Veamos un ejemplo práctico de cómo usar este método:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Geolocation</title>
  <script src="geolocation.js"></script>
</head>
<body>
  <section id="ubicacion">
    <button id="obtener">Obtener mi Ubicación</button>
  </section>
</body>
</html>
```

Listado 9-1. *Documento HTML para la API Geolocation.*

El Listado 9-1 será nuestra plantilla HTML para el resto de este capítulo. Es lo más elemental posible, con tan solo un elemento **<button>** dentro de un elemento **<section>** que vamos a usar para solicitar y mostrar la información retornada por el sistema de ubicación.

```
function iniciar(){
  var boton=document.getElementById('obtener');
  boton.addEventListener('click', obtener, false);
}
function obtener(){
  navigator.geolocation.getCurrentPosition(mostrar);
}
function mostrar(posicion){
  var ubicacion=document.getElementById('ubicacion');
  var datos='';
  datos+='Latitud: '+posicion.coords.latitude+'<br>';
  datos+='Longitud: '+posicion.coords.longitude+'<br>';
  datos+='Exactitud: '+posicion.coords.accuracy+'mts.<br>';
  ubicacion.innerHTML=datos;
}
window.addEventListener('load', iniciar, false);
```

Listado 9-2. *Obteniendo información sobre la localización del usuario.*

Una implementación de la API Geolocation es sencilla: declaramos el método **getCurrentPosition()** y creamos una función que mostrará los valores retornados por el mismo. El método **getCurrentPosition()** es un método del objeto **geolocation**. Este es un nuevo objeto que es parte del objeto **navigator**, un objeto Javascript que fue anteriormente implementado para retornar información acerca del navegador y el sistema. Por lo tanto, para acceder al método **getCurrentPosition()** la sintaxis a usar es **navigator.geolocation.getCurrentPosition(función)**, donde **función** es una función personalizada que recibirá el objeto **Position** y procesará la información que contiene.

En el código del Listado 9-2, llamamos a esta función **mostrar()**. Cuando el método **getCurrentPosition()** es llamado, un nuevo objeto **Position** es creado con la información de la ubicación actual del usuario y es enviado a la función **mostrar()**. Referenciamos este objeto dentro de la función con la variable **posicion**, y luego usamos esta variable para mostrar los datos.

El objeto **Position** tiene dos importantes atributos: **coords** y **timestamp**. En nuestro ejemplo solo usamos **coords** para acceder a la información que queremos mostrar (latitud, longitud y exactitud). Estos valores son grabados en la variable **datos** y luego mostrados en la pantalla como el nuevo contenido del elemento **ubicacion**.

> **Hágalo usted mismo:** Cree archivos con los códigos de los Listados 9-1 y 9-2, suba los archivos a su servidor y luego abra el documento HTML en su navegador. Cuando haga clic en el botón, el navegador le preguntará si desea activar o no el sistema de ubicación geográfica. Si le permite a la aplicación acceder a esta información, entonces su ubicación, incluyendo longitud, latitud y exactitud, será mostrada en pantalla.

getCurrentPosition(ubicación, error)

¿Pero qué ocurre si usted no permite al navegador acceder a información acerca de su ubicación? Agregando un segundo atributo al método **getCurrentPosition()**, con otra función, podremos capturar los errores producidos en el proceso. Uno de esos errores, por supuesto, ocurre cuando el usuario no acepta compartir sus datos.

Junto con el objeto **Position**, el método **getCurrentPosition()** retorna el objeto **PositionError** si un error es detectado. Este objeto es enviado al segundo atributo de **getCurrentPosition()**, y tiene dos atributos internos, **error** y **message**, para proveer el valor y la descripción del error. Los tres posibles errores son representados por las siguientes constantes:

PERMISSION_DENIED (permiso denegado) - valor 1. Este error ocurre cuando el usuario no acepta activar el sistema de ubicación para compartir su información.

POSITION_UNAVAILABLE (ubicación no disponible) - valor 2. Este error ocurre cuando la ubicación del dispositivo no pudo determinarse por ninguno de los sistemas de ubicación disponibles.

TIMEOUT (tiempo excedido) - valor 3. Este error ocurre cuando la ubicación no pudo ser determinada en el período

de tiempo máximo declarado en la configuración.

```
function iniciar(){
  var boton=document.getElementById('obtener');
  boton.addEventListener('click', obtener, false);
}
function obtener(){
  navigator.geolocation.getCurrentPosition(mostrar, errores);
}
function mostrar(posicion){
  var ubicacion=document.getElementById('ubicacion');
  var datos='';
  datos+='Latitud: '+posicion.coords.latitude+'<br>';
  datos+='Longitud: '+posicion.coords.longitude+'<br>';
  datos+='Exactitud: '+posicion.coords.accuracy+'mts.<br>';
  ubicacion.innerHTML=datos;
}
function errores(error){
  alert('Error: '+error.code+' '+error.message);
}
window.addEventListener('load', iniciar, false);
```

Listado 9-3. *Mostrando mensajes de error.*

Los mensajes de error son ofrecidos para uso interno. El propósito es ofrecer un mecanismo para que la aplicación reconozca la situación y proceda de acuerdo al error recibido. En el código del Listado 9-3, agregamos un segundo atributo al método **getCurrentPosition()** (otra función) y creamos la función **errores()** para mostrar la información de los atributos **code** y **message**. El valor de **code** será un número entre 0 y 3 de acuerdo al número de error (listado anteriormente).

El objeto **PositionError** es enviado a la función **errores()** y representado en esta función por la variable **error**. Para propósitos didácticos, usamos un método **alert()** que muestra los datos recibidos, pero usted debería procesar esta información en silencio, si es posible, sin alertar al usuario de nada. Podemos también controlar por errores de forma individual (**error.PERMISSION_DENIED**, por ejemplo) y actuar solo si ese error en particular ocurrió.

getCurrentPosition(ubicación, error, configuración)

El tercer atributo que podemos usar en el método **getCurrentPosition()** es un objeto conteniendo hasta tres posibles propiedades:

enableHighAccuracy Esta es una propiedad booleana para informar al sistema que requerimos de la información más exacta que nos pueda ofrecer. El navegador intentará obtener esta información a través de sistemas como GPS, por ejemplo, para retornar la ubicación exacta del dispositivo. Estos son sistemas que consumen muchos recursos, por lo que su uso debería estar limitado a circunstancias muy específicas. Para evitar consumos innecesarios, el valor por defecto de esta propiedad es **false** (falso).

timeout Esta propiedad indica el tiempo máximo de espera para que la operación finalice. Si la información de la ubicación no es obtenida antes del tiempo indicado, el error **TIMEOUT** es retornado. Su valor es en milisegundos.

maximumAge Las ubicaciones encontradas previamente son almacenadas en un caché en el sistema. Si consideramos apropiado recurrir a la información grabada en lugar de intentar obtenerla desde el sistema (para evitar consumo de recursos o para una respuesta rápida), esta propiedad puede ser declarada con un tiempo límite específico. Si la última ubicación almacenada es más vieja que el valor de este atributo entonces una nueva ubicación es solicitada al sistema. Su valor es en milisegundos.

```
function iniciar(){
  var boton=document.getElementById('obtener');
  boton.addEventListener('click', obtener, false);
}
function obtener(){
  var geoconfig={
    enableHighAccuracy: true,
```

```
    timeout: 10000,
    maximumAge: 60000
  };
  navigator.geolocation.getCurrentPosition(mostrar, errores,
                                                   geoconfig);
}
function mostrar(posicion){
  var ubicacion=document.getElementById('ubicacion');
  var datos='';
  datos+='Latitud: '+posicion.coords.latitude+'<br>';
  datos+='Longitud: '+posicion.coords.longitude+'<br>';
  datos+='Exactitud: '+posicion.coords.accuracy+'mts.<br>';
  ubicacion.innerHTML=datos;
}

function errores(error){
  alert('Error: '+error.code+' '+error.message);
}
window.addEventListener('load', iniciar, false);
```

Listado 9-4. *Configuración del sistema.*

El código del Listado 9-4 intentará obtener la ubicación más exacta posible del dispositivo en no más de 10 segundos, pero solo si no hay una ubicación previa en el caché capturada menos de 60 segundos atrás (si existe una ubicación previa con menos de 60 segundos de antigüedad, éste será el objeto **Position** retornado).

El objeto conteniendo los valores de configuración fue creado primero y luego referenciado desde el método **getCurrentPosition()**. Nada cambió en el resto del código. La función **mostrar()** mostrará la información en la pantalla independiente-mente de su origen (si proviene del caché o es nueva).

Conceptos básicos: Javascript provee diferentes formas de construir un objeto. Por propósitos de claridad, elegimos crear el objeto primero, almacenarlo en la variable **geoconfig** y luego usar esta referencia en el método **getCurrentPosition()**. Sin embargo, podríamos haber insertado el objeto directamente en el método como un atributo. En aplicaciones pequeñas, objetos normalmente pueden ser evitados, pero no es el caso de códigos más complejos. Para aprender más sobre objetos y programación orientada a objetos en Javascript, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Con el último código, podemos apreciar el propósito real de la API Geolocation y cuál fue la intención de sus desarrolladores. Las funciones más efectivas y prácticas están orientadas hacia dispositivos móviles. El valor **true** (verdadero) para la propiedad **enableHighAccuracy**, por ejemplo, le solicitará al navegador usar sistemas como GPS para obtener la ubicación más exacta posible (un sistema casi exclusivo de dispositivos móviles), y los métodos **watchPosition()** y **clearWatch()**, que veremos a continuación, trabajan sobre ubicaciones actualizadas constantemente, algo solo posible, por supuesto, cuando el dispositivo que está accediendo la aplicación es móvil (y se está moviendo). Esto trae a la luz dos asuntos importantes. Primero, la mayoría de nuestros códigos tendrán que ser probados en un dispositivo móvil para saber exactamente cómo trabajan en una situación real. Y segundo, deberemos ser responsables con el uso de esta API. GPS y otros sistemas de localización consumen muchos recursos y en la mayoría de los casos pueden acabar pronto con la batería del dispositivo si no somos cautelosos.

Con respecto al primer punto, disponemos de una alternativa. Simplemente visite el enlace *dev.w3.org/geo/api/test-suite/* y lea acerca de cómo experimentar y probar Geolocation API. Con respecto al segundo punto, solo un consejo: configure la propiedad **enableHighAccuracy** como **true** solo cuando es estrictamente necesario, y no abuse de esta posibilidad.

watchPosition(ubicación, error, configuración)

Similar a **getCurrentPosition()**, el método **watchPosition()** recibe tres atributos y realiza la misma tarea: obtener la ubicación del dispositivo que está accediendo a la aplicación. La única diferencia es que el primero realiza una única operación, mientras que **watchPosition()** ofrece nuevos datos cada vez que la ubicación cambia. Este método vigilará todo el tiempo la ubicación y enviará información a la función correspondiente cuando se detecte una nueva ubicación, a menos que cancelemos el proceso con el método **clearWatch()**.

Este es un ejemplo de cómo implementar el método **watchPosition()** basado en códigos previos:

```
function iniciar(){
  var boton=document.getElementById('obtener');
```

```
  boton.addEventListener('click', obtener, false);
}
function obtener(){
  var geoconfig={
    enableHighAccuracy: true,
    maximumAge: 60000
  };
  control=navigator.geolocation.watchPosition(mostrar, errores,
                                                     geoconfig);
}
function mostrar(posicion){
  var ubicacion=document.getElementById('ubicacion');
  var datos='';
  datos+='Latitud: '+posicion.coords.latitude+'<br>';
  datos+='Longitud: '+posicion.coords.longitude+'<br>';
  datos+='Exactitud: '+posicion.coords.accuracy+'mts.<br>';
  ubicacion.innerHTML=datos;
}
function errores(error){
  alert('Error: '+error.code+' '+error.message);
}
window.addEventListener('load', iniciar, false);
```

Listado 9-5. *Probando el método* `watchPosition()`.

No notará ningún cambio en un ordenador de escritorio usando este código, pero en un dispositivo móvil nueva información será mostrada cada vez que haya una modificación en la ubicación del dispositivo. El atributo **`maximumAge`** determina qué tan seguido la información será enviada a la función **`mostrar()`**. Si la nueva ubicación es obtenida 60 segundos (60000 milisegundos) luego de la anterior, entonces será mostrada, en caso contrario la función **`mostrar()`** no será llamada.

Note que el valor retornado por el método **`watchPosition()`** fue almacenado en la variable **`control`**. Esta variable es como un identificador de la operación. Si más adelante queremos cancelar el proceso de vigilancia, solo debemos ejecutar la línea **`clearWatch(control)`** y **`watchPosition()`** dejará de actualizar la información.

Si ejecuta este código en un ordenador de escritorio, el método **`watchPosition()`** funcionará como el anterior estudiado **`getCurrentPosition()`**; la información no será actualizada. La función **`mostrar()`** es solo llamada cuando la ubicación cambia.

Usos prácticos con Google Maps

Hasta el momento hemos mostrado la información sobre la ubicación exactamente como la recibimos. Sin embargo, estos valores normalmente no significan nada para la gente común. La mayoría de nosotros no podemos inmediatamente decir cuál es nuestra actual ubicación en valores de latitud y longitud, y mucho menos identificar a partir de estos valores una ubicación en el mundo. Disponemos de dos alternativas: usar esta información internamente para calcular posiciones, distancias y otros valores que nos permitirán ofrecer resultados específicos a nuestros usuarios (como productos o servicios en el área), o podemos ofrecer la información obtenida por medio de la API Geolocation en un medio mucho más comprensible. ¿Y qué más comprensible que un mapa para representar una ubicación geográfica?

Más atrás en este libro hablamos acerca de la API Google Maps. Esta es una API Javascript externa, provista por Google, que nada tiene que ver con HTML5 pero es incluida extraoficialmente dentro de la especificación y es ampliamente utilizada en sitios webs modernos estos días. Ofrece una variedad de alternativas para trabajar con mapas interactivos e incluso vistas reales de lugares muy específicos a través de la tecnología StreetView.

Vamos a mostrar un ejemplo simple de utilización aprovechando una parte de la API llamada Static Maps API. Con esta API específica, solo necesitamos construir una URL con la información de la ubicación para obtener en respuesta la imagen de un mapa con el área seleccionada.

```
function iniciar(){
  var boton=document.getElementById('obtener');
  boton.addEventListener('click', obtener, false);
}
function obtener(){
```

```
    navigator.geolocation.getCurrentPosition(mostrar, errores);
  }
  function mostrar(posicion){
    var ubicacion=document.getElementById('ubicacion');
    var mapurl='http://maps.google.com/maps/api/staticmap?center='+
       posicion.coords.latitude+','+posicion.coords.longitude+'&zoom=
     12&size=400x400&sensor=false&markers='+posicion.coords.latitude+
                                     ','+posicion.coords.longitude;
    ubicacion.innerHTML='<img src="'+mapurl+'">';
  }
  function errores(error){
    alert('Error: '+error.code+' '+error.message);
  }
  window.addEventListener('load', iniciar, false);
```

Listado 9-6. *Representando la ubicación en un mapa.*

El código es simple. Usamos el método **getCurrentPosition()** y enviamos la información a la función **mostrar()** como siempre, pero ahora en esta función los valores del objeto **Position** son agregados a una URL de Google y luego la dirección obtenida es insertada como la fuente de un elemento **<img>** para mostrar el mapa en pantalla.

> **Hágalo usted mismo:** Pruebe el código del Listado 9-6 en su navegador usando la plantilla del Listado 9-1. Cambie los valores de los atributos **zoom** y **size** en la URL para modificar el mapa retornado por la API. Visite la página de Google Maps API para estudiar las diferentes alternativas provistas por esta API: *code.google.com/ apis/maps/.*

9.2 Referencia rápida

Determinar la ubicación física del usuario se ha vuelto crítico en aplicaciones web modernas. El reciente éxito de los dispositivos móviles ofrece nuevas posibilidades para crear aplicaciones que aprovechan esta información.

Métodos

La API Geolocation provee tres métodos para obtener la ubicación de un dispositivo:

getCurrentPosition(ubicación, error, configuración) Este método retorna información sobre la ubicación del dispositivo que está accediendo a la aplicación. El primer atributo es una función destinada a procesar la información, el segundo atributo es otra función para procesamiento de errores, y el tercer atributo es un objeto con valores de configuración (vea Objeto Configuración debajo).

watchPosition(ubicación, error, configuración) Este método retorna información sobre la ubicación del dispositivo que está accediendo a la aplicación cada vez que la ubicación cambia. El primer atributo es una función destinada a procesar la información, el segundo atributo es otra función para procesamiento de errores, y el tercer atributo es un objeto con valores de configuración (vea Objeto Configuración debajo).

clearWatch(id) Este método cancela el proceso que ha sido empezado por el método **`watchPosition()`**. El atributo **`id`** es el valor de identificación retornado por el método **`watchPosition()`** cuando es llamado.

Objetos

Los métodos **`getCurrentPosition()`** y **`watchPosition()`** generan dos objetos para comunicar la información retornada por el sistema de ubicación y el estado de la operación.

Objeto Position Este objeto es generado para contener la información acerca de la ubicación detectada. Tiene dos atributos: **`coords`** y **`timestamp`**.

coords Este es un atributo del objeto **`Position`**. Tiene siete atributos internos para retornar la información de la ubicación: **`latitude`** (latitud), **`longitude`** (longitud), **`altitude`** (altitud en metros), **`accuracy`** (exactitud en metros), **`altitudeAccuracy`** (exactitud de la altitud en metros), **`heading`** (dirección en grados) y **`speed`** (velocidad en metros por segundo).

timestamp Este es un atributo del objeto **`Position`**. Retorna el momento en el que la ubicación fue detectada.

Objeto PositionError Este objeto es generado cuando un error ocurre. Ofrece dos atributos generales con el valor y el mensaje del error, y tres valores específicos para identificación de errores individuales (listados debajo).

message Este es un atributo del objeto **`PositionError`**. Retorna un mensaje describiendo el error detectado.

error Este es un atributo del objeto **`PositionError`**. Contiene el valor del error detectado. Los posibles valores son listados debajo:

PERMISSION_DENIED (permiso denegado) - valor 1 en el atributo **`error`**. Esta constante es **`true`** (verdadero) cuando el usuario no permite a la aplicación acceder a la información sobre su ubicación.

POSITION_UNAVAILABLE (ubicación no disponible) - valor 2 en el atributo **`error`**. Esta constante es **`true`** (verdadero) cuando la ubicación del dispositivo no puede ser determinada.

TIMEOUT (tiempo excedido) - valor 3 en el atributo **`error`**. Esta constante es **`true`** (verdadero) cuando la ubicación no puede ser determinada antes del periodo de tiempo declarado en la configuración.

El siguiente objeto es requerido por los métodos **`getCurrentPosition()`** y **`watchPosition()`** para propósitos de configuración.

Objeto Configuración Este objeto provee valores de configuración correspondientes para los métodos **`getCurrentPosition()`** y **`watchPosition()`**.

enableHighAccuracy Esta es una de las posibles propiedades del Objeto Configuración. Si es declarada como **`true`** (verdadero), le solicitará al navegador obtener la ubicación más precisa posible.

timeout Esta es una de las propiedades del Objeto Configuración. Indica el máximo tiempo disponible que tiene la operación para realizarse.

maximumAge Esta es una de las propiedades del Objeto Configuración. Indica por cuánto tiempo la última

ubicación detectada será válida.

Capítulo 10
API Web Storage

10.1 Dos sistemas de almacenamiento

La Web fue primero pensada como una forma de mostrar información, solo mostrarla. El procesamiento de información comenzó luego, primero con aplicaciones del lado del servidor y más tarde, de forma bastante ineficiente, a través de pequeños códigos y complementos (plug-ins) ejecutados en el ordenador del usuario. Sin embargo, la esencia de la Web siguió siendo básicamente la misma: la información era preparada en el servidor y luego mostrada a los usuarios. El trabajo duro se desarrollaba casi completamente del lado del servidor porque el sistema no aprovechaba los recursos en los ordenadores de los usuarios.

HTML5 equilibra esta situación. Justificada por las particulares características de los dispositivos móviles, el surgimiento de los sistemas de computación en la nube, y la necesidad de estandarizar tecnologías e innovaciones introducidas por plug-ins a través de los últimos años, la especificación de HTML5 incluye herramientas que hacen posible construir y ejecutar aplicaciones completamente funcionales en el ordenador del usuario, incluso cuando no existe conexión a la red disponible.

Una de las características más necesitadas en cualquier aplicación es la posibilidad de almacenar datos para disponer de ellos cuando sean necesarios, pero no existía aún un mecanismo efectivo para este fin. Las llamadas “Cookies” (archivos de texto almacenados en el ordenador del usuario) fueron usadas por años para preservar información, pero debido a su naturaleza se encontraron siempre limitadas a pequeñas cadenas de texto, lo que las hacía útiles solo en determinadas circunstancias.

La API Web Storage es básicamente una mejora de las Cookies. Esta API nos permite almacenar datos en el disco duro del usuario y utilizarlos luego del mismo modo que lo haría una aplicación de escritorio. El proceso de almacenamiento provisto por esta API puede ser utilizado en dos situaciones particulares: cuando la información tiene que estar disponible solo durante la sesión en uso, y cuando tiene que ser preservada todo el tiempo que el usuario desee. Para hacer estos métodos más claros y comprensibles para los desarrolladores, la API fue dividida en dos partes llamadas **`sessionStorage`** y **`localStorage`**.

sessionStorage Este es un mecanismo de almacenamiento que conservará los datos disponible solo durante la duración de la sesión de una página. De hecho, a diferencia de sesiones reales, la información almacenada a través de este mecanismo es solo accesible desde una única ventana o pestaña y es preservada hasta que la ventana es cerrada. La especificación aún nombra “sesiones” debido a que la información es preservada incluso cuando la ventana es actualizada o una nueva página desde el mismo sitio web es cargada.

localStorage Este mecanismo trabaja de forma similar a un sistema de almacenamiento para aplicaciones de escritorio. Los datos son grabados de forma permanente y se encuentran siempre disponibles para la aplicación que los creó.

Ambos mecanismos trabajan a través de la misma interface, compartiendo los mismos métodos y propiedades. Y ambos son dependientes del origen, lo que quiere decir que la información está disponible solo a través del sitio web o la aplicación que los creó. Cada sitio web tendrá designado su propio espacio de almacenamiento que durará hasta que la ventana es cerrada o será permanente, de acuerdo al mecanismo utilizado.

La API claramente diferencia datos temporarios de permanentes, facilitando la construcción de pequeñas aplicaciones que necesitan preservar solo unas cadenas de texto como referencia temporaria (por ejemplo, carros de compra) o aplicaciones más grandes y complejas que necesitan almacenar documentos completos por todo el tiempo que sea necesario.

IMPORTANTE: Muchos navegadores solo trabajan de forma adecuada con esta API cuando la fuente es un servidor real. Para probar los siguientes códigos, le recomendamos que primero suba los archivos a su servidor.

10.2 La sessionStorage

Esta parte de la API, **`sessionStorage`**, es como un reemplazo para las Cookies de sesión. Las Cookies, así como **`sessionStorage`**, mantienen los datos disponibles durante un período específico de tiempo, pero mientras las Cookies de sesión usan el navegador como referencia, **`sessionStorage`** usa solo una simple ventana o pestaña. Esto significa que las Cookies creadas para una sesión estarán disponibles mientras el navegador continúe abierto, mientras que los datos creados con **`sessionStorage`** estarán solo disponibles mientras la ventana que los creó no es cerrada.

Implementación de un sistema de almacenamiento de datos

Debido a que ambos sistemas, **`sessionStorage`** y **`localStorage`**, trabajan con la misma interface, vamos a necesitar solo un documento HTML y un simple formulario para probar los códigos y experimentar con esta API:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Web Storage API</title>
  <link rel="stylesheet" href="storage.css">
  <script src="storage.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Clave:<br><input type="text" name="clave" id="clave"></p>
      <p>Valor:<br><textarea name="text" id="texto"></textarea></p>
      <p><input type="button" name="grabar" id="grabar"
                                              value="Grabar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 10-1. *Plantilla para la API Storage*

También crearemos un grupo de reglas de estilo simples para dar forma a la página y diferenciar el área del formulario de la caja donde los datos serán mostrados y listados:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#clave, #texto{
  width: 200px;
}
#cajadatos > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 10-2. *Estilos para nuestra plantilla.*

Hágalo usted mismo: Cree un archivo HTML con el código del Listado 10-1 y un archivo CSS llamado **`storage.css`** con los estilos del Listado 10-2. También necesitará crear un archivo llamado **`storage.js`** para grabar y probar los códigos Javascript presentados a continuación.

Creando datos

Ambos, **`sessionStorage`** y **`localStorage`**, almacenan datos como ítems. Los ítems están formados por un par clave/valor, y cada valor será convertido en una cadena de texto antes de ser almacenado. Piense en ítems como si fueran variables, con un nombre y un valor, que pueden ser creadas, modificadas o eliminadas.

Existen dos nuevos métodos específicos de esta API incluidos para crear y leer un valor en el espacio de almacenamiento:

setItem(clave, valor) Este es el método que tenemos que llamar para crear un ítem. El ítem será creado con una clave y un valor de acuerdo a los atributos especificados. Si ya existe un ítem con la misma clave, será actualizado al nuevo valor, por lo que este método puede utilizarse también para modificar datos previos.

getItem(clave) Para obtener el valor de un ítem, debemos llamar a este método especificando la clave del ítem que queremos leer. La clave en este caso es la misma que declaramos cuando creamos al ítem con **`setItem()`**.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
  var valor=document.getElementById('texto').value;
  sessionStorage.setItem(clave,valor);

  mostrar(clave);
}
function mostrar(clave){
  var cajadatos=document.getElementById('cajadatos');
  var valor=sessionStorage.getItem(clave);
  cajadatos.innerHTML='<div>'+clave+' - '+valor+'</div>';
}
window.addEventListener('load', iniciar, false);
```

Listado 10-3. *Almacenando y leyendo datos.*

El proceso es extremadamente simple. Los métodos son parte de **`sessionStorage`** y son llamados con la sintaxis **`sessionStorage.setItem()`**. En el código del Listado 10-3, la función **`nuevoitem()`** es ejecutada cada vez que el usuario hace clic en el botón del formulario. Esta función crea un ítem con la información insertada en los campos del formulario y luego llama a la función **`mostrar()`**. Esta última función lee el ítem de acuerdo a la clave recibida usando el método **`getItem()`** y muestra su valor en la pantalla.

Además de estos métodos, la API también ofrece una sintaxis abreviada para crear y leer ítems desde el espacio de almacenamiento. Podemos usar la clave del ítem como una propiedad y acceder a su valor de esta manera.

Este método usa en realidad dos tipos de sintaxis diferentes de acuerdo al tipo de información que estamos usando para crear el ítem. Podemos encerrar una variable representando la clave entre corchetes (por ejemplo, **`sessionStorage[clave]=valor`**) o podemos usar directamente el nombre de la propiedad (por ejemplo, **`sessionStorage.miitem=valor)`**.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
```

```
    var valor=document.getElementById('texto').value;
    sessionStorage[clave]=valor;

    mostrar(clave);
  }
  function mostrar(clave){
    var cajadatos=document.getElementById('cajadatos');
    var valor=sessionStorage[clave];
    cajadatos.innerHTML='<div>'+clave+' - '+valor+'</div>';
  }
  window.addEventListener('load', iniciar, false);
```

Listado 10-4. *Usando un atajo para trabajar con ítems.*

Leyendo datos

El anterior ejemplo solo lee el último ítem grabado. Vamos a mejorar este código aprovechando más métodos y propiedades provistos por la API con el propósito de manipular ítems:

length Esta propiedad retorna el número de ítems guardados por esta aplicación en el espacio de almacenamiento. Trabaja exactamente como la propiedad **`length`** usada normalmente en Javascript para procesar arrays, y es útil para lecturas secuenciales.

key(índice) Los ítems son almacenados secuencialmente, enumerados con un índice automático que comienzo por 0. Con este método podemos leer un ítem específico o crear un bucle para obtener toda la información almacenada.

```
  function iniciar(){
    var boton=document.getElementById('grabar');
    boton.addEventListener('click', nuevoitem, false);
    mostrar();
  }
  function nuevoitem(){
    var clave=document.getElementById('clave').value;
    var valor=document.getElementById('texto').value;

    sessionStorage.setItem(clave,valor);
    mostrar();
    document.getElementById('clave').value='';
    document.getElementById('texto').value='';
  }
  function mostrar(){
    var cajadatos=document.getElementById('cajadatos');
    cajadatos.innerHTML='';
    for(var f=0;f<sessionStorage.length;f++){
      var clave=sessionStorage.key(f);
      var valor=sessionStorage.getItem(clave);
      cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
    }
  }
  window.addEventListener('load', iniciar, false);
```

Listado 10-5. *Lstando ítems.*

El propósito del código en el Listado 10-5 es mostrar un listado completo de los ítems en la caja derecha de la pantalla. La función **`mostrar()`** fue mejorada usando la propiedad **`length`** y el método **`key()`**. Creamos un bucle **`for`** que va desde 0 al número de ítems que existen en el espacio de almacenamiento. Dentro del bucle, el método **`key()`** retornará la clave que nosotros definimos para cada ítem. Por ejemplo, si el ítem en la posición 0 del espacio de almacenamiento fue creado con la clave “miitem”, el código **`sessionStorage.key(0)`** retornará el valor “miitem”. Llamando a este método desde un bucle podemos listar todos los ítems en la pantalla con sus correspondientes claves y valores.

La función **`mostrar()`** es llamada desde la función **`iniciar()`** tan pronto como la aplicación es ejecutada. De este

modo podremos ver desde el comienzo los ítems que fueron grabados previamente en el espacio de almacenamiento.

Hágalo usted mismo: Aproveche los conceptos estudiados con la API Forms en el Capítulo 6 para controlar la validez de los campos del formulario y no permitir la inserción de ítems vacíos o inválidos.

Eliminando datos

Los ítems pueden ser creados, leídos y, por supuesto, eliminados. Es hora de ver cómo eliminar un ítem. La API ofrece dos métodos para este propósito:

removeItem(clave) Este método eliminará un ítem individual. La clave para identificar el ítem es la misma declarada cuando el ítem fue creado con el método **`setItem()`**.

clear() Este método vaciará el espacio de almacenamiento. Todos los ítems serán eliminados.

```
function iniciar(){
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', nuevoitem, false);
  mostrar();
}
function nuevoitem(){
  var clave=document.getElementById('clave').value;
  var valor=document.getElementById('texto').value;

  sessionStorage.setItem(clave,valor);
  mostrar();
  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
}
function mostrar(){
  var cajadatos=document.getElementById('cajadatos');
    cajadatos.innerHTML='<div><button
       onclick="eliminarTodo()">Eliminar Todo</button></div>';
  for(var f=0;f<sessionStorage.length;f++){
    var clave=sessionStorage.key(f);
    var valor=sessionStorage.getItem(clave);
    cajadatos.innerHTML+='<div>'+clave+' - '+valor+'<br><button
      onclick="eliminar(\''+clave+'\')">Eliminar</button></div>';
  }
}
function eliminar(clave){
  if(confirm('Está Seguro?')){
    sessionStorage.removeItem(clave);
    mostrar();
  }
}
function eliminarTodo(){
  if(confirm('Está Seguro?')){
    sessionStorage.clear();
    mostrar();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 10-6. *Eliminando ítems.*

Las funciones **`iniciar()`** y **`nuevoitem()`** en el Listado 10-6 son las mismas de códigos previos. Solo la función **`mostrar()`** cambia para incorporar el manejador de eventos **`onclick`** y llamar a las funciones que eliminarán un ítem individual o vaciarán el espacio de almacenamiento. La lista de ítems presentada en pantalla es construida de la misma manera que antes, pero esta vez un botón “Eliminar” es agregado junto a cada ítem para poder eliminarlo. Un botón para eliminar todos los ítems juntos también fue agregado en la parte superior.

Las funciones **`eliminar()`** y **`eliminarTodo()`** se encargan de eliminar el ítem seleccionado o limpiar el espacio de almacenamiento, respectivamente. Cada función llama a la función **`mostrar()`** al final para actualizar la lista de ítems en pantalla.

Hágalo usted mismo: Con el código del Listado 10-6, podrá estudiar cómo la información es procesada por **`sessionStorage`**. Abra la plantilla del Listado 10-1 en su navegador, cree nuevos ítems y luego abra la plantilla en una nueva ventana. La información en cada ventana es diferente. La vieja ventana mantendrá su información disponible y el espacio de almacenamiento de la nueva ventana estará vacío. A diferencia de otros sistemas (como Cookies de sesiones), para **`sessionStorage`** cada ventana es considerada una instancia diferente de la aplicación y la información de la sesión no se propaga entre ellas.

El sistema **`sessionStorage`** preserva los datos creados en una ventana solo hasta que esa ventana es cerrada. Es útil para controlar carros de compra o cualquier otra aplicación que requiere acceso a datos por períodos cortos de tiempo.

10.3 La localStorage

Disponer de un sistema confiable para almacenar datos durante la sesión de una ventana puede ser extremadamente útil en algunas circunstancias, pero cuando intentamos simular poderosas aplicaciones de escritorio en la web, un sistema de almacenamiento temporario no es suficiente.

Para cubrir este aspecto, Storage API ofrece un segundo sistema que reservará un espacio de almacenamiento para cada aplicación (cada origen) y mantendrá la información disponible permanentemente. Con **localStorage**, finalmente podemos grabar largas cantidades de datos y dejar que el usuario decida si la información es útil y debe ser conservada o no.

El sistema usa la misma interface que **sessionStorage**, debido a esto cada método y propiedad estudiado hasta el momento en este capítulo son también disponibles para **localStorage**. Solo la substitución del prefijo **session** por **local** es requerida para preparar los códigos.

```
   function iniciar(){
     var boton=document.getElementById('grabar');
     boton.addEventListener('click', nuevoitem, false);
     mostrar();
   }

function nuevoitem(){
     var clave=document.getElementById('clave').value;
     var valor=document.getElementById('texto').value;

     localStorage.setItem(clave,valor);
     mostrar();
     document.getElementById('clave').value='';
     document.getElementById('texto').value='';
   }
   function mostrar(){
     var cajadatos=document.getElementById('cajadatos');
     cajadatos.innerHTML='';
     for(var f=0;f<localStorage.length;f++){
       var clave=localStorage.key(f);
       var valor=localStorage.getItem(clave);
       cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
     }
   }
   window.addEventListener('load', iniciar, false);
```

Listado 10-7. *Usando* `localStorage`.

En el Listado 10-7, simplemente reemplazamos **sessionStorage** por **localStorage** en el código de uno de los ejemplos anteriores. Ahora, cada ítem creado será preservado a través de diferentes ventanas e incluso luego de que todas las ventanas del navegador son cerradas.

> **Hágalo usted mismo:** Usando la plantilla del Listado 10-1, pruebe el código del Listado 10-7. Este código creará un nuevo ítem con la información insertada en el formulario y automáticamente listará todos los ítems disponibles en el espacio de almacenamiento reservado para esta aplicación. Cierre el navegador y abra el archivo HTML nuevamente. La información es preservada, por lo que podrá ver aún en pantalla todos los ítems ingresados previamente.

Evento storage

Debido a que **localStorage** hace que la información esté disponible en cada ventana donde la aplicación fue cargada, surgen al menos dos problemas: debemos resolver cómo estas ventanas se comunicarán entre sí y cómo haremos para mantener la información actualizada en cada una de ellas. En respuesta a ambos problemas, la especificación incluye el evento **storage**.

storage Este evento será disparado por la ventana cada vez que un cambio ocurra en el espacio de almacenamiento. Puede ser usado para informar a cada ventana abierta con la misma aplicación que los datos han cambiado en el espacio de almacenamiento y que se debe hacer algo al respecto.

```
function iniciar(){
     var boton=document.getElementById('grabar');
     boton.addEventListener('click', nuevoitem, false);
     window.addEventListener("storage", mostrar, false);

     mostrar();
   }
   function nuevoitem(){
     var clave=document.getElementById('clave').value;
     var valor=document.getElementById('texto').value;

     localStorage.setItem(clave,valor);
     mostrar();
     document.getElementById('clave').value='';
     document.getElementById('texto').value='';
   }
   function mostrar(){
     var cajadatos=document.getElementById('cajadatos');
     cajadatos.innerHTML='';
     for(var f=0;f<localStorage.length;f++){
       var clave=localStorage.key(f);
       var valor=localStorage.getItem(clave);
       cajadatos.innerHTML+='<div>'+clave+' - '+valor+'</div>';
     }
   }
   window.addEventListener('load', iniciar, false);
```

Listado 10-8. *Escuchando al evento* `storage` *para mantener la lista de ítems actualizada.*

Solo necesitamos comenzar a escuchar al evento **`storage`** en la función **`iniciar()`** del código 10-8 para ejecutar la función **`mostrar()`** en cada ventana siempre que un ítem es creado, modificado o eliminado. Ahora, si algo cambia en una ventana, el cambio será mostrado automáticamente en el resto de las ventanas que están ejecutando la misma aplicación.

Espacio de almacenamiento

La información almacenada por **`localStorage`** será permanente a menos que el usuario decida que ya no la necesita. Esto significa que el espacio físico en el disco duro ocupado por esta información probablemente crecerá cada vez que la aplicación sea usada. Hasta este momento, la especificación de HTML5 recomienda a los fabricantes de navegadores que reserven un mínimo de 5 megabytes para cada origen (cada sitio web o aplicación).

Esta es solo una recomendación que probablemente cambiará dramáticamente en los próximos años. Algunos navegadores están consultando al usuario si expandir o no el espacio disponible cuando la aplicación lo necesita, pero usted debería ser consciente de esta limitación y tenerla en cuenta a la hora de desarrollar sus aplicaciones.

> **IMPORTANTE:** Muchos navegadores solo trabajan de forma adecuada con esta API cuando la fuente es un servidor real. Para probar los siguientes códigos, le recomendamos que primero suba los archivos a su servidor.

10.4 Referencia rápida

Con la ayuda de la API Web Storage, ahora las aplicaciones web pueden ofrecer un espacio de almacenamiento. Usando un par clave/valor, la información es almacenada en el ordenador del usuario para un rápido acceso o para trabajar desconectado de la red.

Tipo de almacenamiento

Dos mecanismos diferentes son ofrecidos para almacenar datos:

sessionStorage Este mecanismo mantiene la información almacenada solo disponible para una simple ventana y solo hasta que la ventana es cerrada.

localStorage Este mecanismo almacena datos de forma permanente. Estos datos son compartidos por todas las ventanas que están ejecutando la misma aplicación y estarán disponibles a menos que el usuario decida que ya no los necesita.

Métodos

La API incluye una interface común para cada mecanismo que cuenta con nuevos métodos, propiedades y eventos:

setItem(clave, valor) Este método crea un nuevo ítem que es almacenado en el espacio de almacenamiento reservado para la aplicación. El ítem está compuesto por un par clave/valor creado a partir de los atributos `clave` y `valor`.

getItem(clave) Este método lee el contenido de un ítem identificado por la clave especificada por el atributo `clave`. El valor de esta clave debe ser el mismo usado cuando el ítem fue creado con el método `setItem()`.

key(índice) Este método retorna la clave del ítem encontrado en la posición especificada por el atributo `índice` dentro del espacio de almacenamiento.

removeItem(clave) Este método elimina un ítem con la clave especificada por el atributo `clave`. El valor de esta clave debe ser el mismo usado cuando el ítem fue creado por el método `setItem()`.

clear() - Este método elimina todos los ítems en el espacio de almacenamiento reservado para la aplicación.

Propiedades

length Esta propiedad retorna el número de ítems disponibles en el espacio de almacenamiento reservado para la aplicación.

Eventos

storage Este evento es disparado cada vez que un cambio se produce en el espacio de almacenamiento reservado para la aplicación.

Capítulo 11
API IndexedDB

11.1 Una API de bajo nivel

La API estudiada en el capítulo anterior es útil para almacenamiento de pequeñas cantidades de datos, pero cuando se trata de grandes cantidades de datos estructurados debemos recurrir a un sistema de base de datos. La API IndexedDB es la solución provista por HTML5 a este respecto.

IndexedDB es un sistema de base de datos destinado a almacenar información indexada en el ordenador del usuario. Fue desarrollada como una API de bajo nivel con la intención de permitir un amplio espectro de usos. Esto la convierte en una de las API más poderosa de todas, pero también una de las más complejas. El objetivo fue proveer la estructura más básica posible para permitir a los desarrolladores construir librerías y crear interfaces de alto nivel para satisfacer necesidades específicas. En una API de bajo nivel como esta tenemos que hacernos cargo de cada aspecto y controlar las condiciones de cada proceso en toda operación realizada. El resultado es una API a la que la mayoría de los desarrolladores tardará en acostumbrarse y probablemente utilizará de forma indirecta a través de otras librerías populares construidas sobre ella que seguramente surgirán en un futuro cercano.

La estructura propuesta por IndexedDB es también diferente de SQL u otros sistemas de base de datos populares. La información es almacenada en la base de datos como objetos (registros) dentro de lo que es llamado Almacenes de Objetos (tablas). Los Almacenes de Objetos no tienen una estructura específica, solo un nombre e índices para poder encontrar los objetos en su interior. Estos objetos tampoco tienen una estructura definida, pueden ser diferentes unos de otros y tan complejos como necesitemos. La única condición para los objetos es que contengan al menos una propiedad declarada como índice para que sea posible encontrarlos en el Almacén de Objetos.

Base de datos

La base de datos misma es simple. Debido a que cada base de datos es asociada con un ordenador y un sitio web o aplicación, no existen usuarios para agregar o restricciones de acceso que debamos considerar. Solo necesitamos especificar el nombre y la versión, y la base de datos estará lista.

La interface declarada en la especificación para esta API provee el atributo **`indexedDB`** y el método **`open()`** para crear la base de datos. Este método retorna un objeto sobre el cual dos eventos serán disparados para indicar el éxito o los errores surgidos durante el proceso de creación de la base de datos.

El segundo aspecto que debemos considerar para crear o abrir una base de datos es la versión. La API requiere que una versión sea asignada a la base de datos. Esto es para preparar al sistema para futuras migraciones. Cuando tenemos que actualizar la estructura de una base de datos en el lado del servidor para agregar más tablas o índices, normalmente detenemos el servidor, migramos la información hacia la nueva estructura y luego encendemos el servidor nuevamente. Sin embargo, en este caso la información es contenida del lado del cliente y, por supuesto, el ordenador del usuario no puede ser apagado. Como resultado, la versión de la base de datos debe ser cambiada y luego la información migrada desde la vieja a la nueva versión.

Para trabajar con versiones de bases de datos, la API ofrece la propiedad **`version`** y el método **`setVersion()`**. La propiedad retorna el valor de la versión actual y el método asigna un nuevo valor de versión a la base de datos en uso. Este valor puede ser numérico o cualquier cadena de texto que se nos ocurra.

Objetos y Almacenes de Objetos

Lo que solemos llamar registros, en IndexedDB son llamados objetos. Estos objetos incluyen propiedades para almacenar e identificar valores. La cantidad de propiedades y cómo los objetos son estructurados es irrelevante. Solo deben incluir al menos una propiedad declarada como índice para poder encontrarlos dentro del Almacén de Objetos.

Los Almacenes de Objetos (tablas) tampoco tienen una estructura específica. Solo el nombre y uno o más índices deben ser declarados en el momento de su creación para poder encontrar objetos en su interior más tarde.



Figura 11-1. *Objetos con diferentes propiedades almacenados en un Almacén de Objetos.*

Como podemos ver en la Figura 11-1, un Almacén de Objetos contiene diversos objetos con diferentes propiedades. Algunos objetos tienen una propiedad **`DVD`**, otros tienen una propiedad **`Libro`**, etc… Cada uno tiene su propia estructura, pero todos deberán tener al menos una propiedad declarada como índice para poder ser encontrados. En el ejemplo de la Figura 11-1, este índice podría ser la propiedad **`Id`**.

Para trabajar con objetos y Almacenes de Objetos solo necesitamos crear el Almacén de Objetos, declarar las propiedades que serán usadas como índices y luego comenzar a almacenar objetos en este almacén. No necesitamos pensar acerca de la estructura o el contenido de los objetos en este momento, solo considerar los índices que vamos a utilizar para encontrarlos más adelante en el almacén.

La API provee varios métodos para manipular Almacenes de Objetos:

createObjectStore(nombre, clave, autoIncremento) Este método crea un nuevo Almacén de Objetos con el nombre y la configuración declarada por sus atributos. El atributo **`nombre`** es obligatorio. El atributo **`clave`** declarará un índice común para todos los objetos. Y el atributo **`autoIncremento`** es un valor booleano que determina si el Almacén de Objetos tendrá un generador de claves automático.

objectStore(nombre) Para acceder a los objetos en un Almacén de Objetos, una transacción debe ser iniciada y el Almacén de Objetos debe ser abierto para esa transacción. Este método abre el Almacén de Objetos con el nombre declarado por el atributo **`nombre`**.

deleteObjectStore(nombre) Este método destruye un Almacén de Objetos con el nombre declarado por el atributo **`nombre`**.

Los métodos **`createObjectStore()`** y **`deleteObjectStore()`**, así como otros métodos responsables de la configuración de la base de datos, pueden solo ser aplicados cuando la base de datos es creada o mejorada en una nueva versión.

Índices

Para encontrar objetos en un Almacén de Objetos necesitamos declarar algunas propiedades de estos objetos como índices. Una forma fácil de hacerlo es declarar el atributo **`clave`** en el método **`createObjectStore()`**. La propiedad declarada como **`clave`** será un índice común para cada objeto almacenado en ese Almacén de Objetos particular. Cuando declaramos el atributo **`clave,`** esta propiedad debe estar presente en todos los objetos.

Además del atributo **`clave`** podemos declarar todos los índices que necesitemos para un Almacén de Objetos usando métodos especiales provistos para este propósito:

createIndex(nombre, propiedad, único) Este método crea un índice para un Almacén de Objetos específico. El atributo **`nombre`** es un nombre con el que identificar al índice, el atributo **`propiedad`** es la propiedad que será usada como índice, y **`unique`** es un valor booleano que indica si existe la posibilidad de que dos o más objetos compartan el mismo valor para este índice.

index(nombre) Para usar un índice primero tenemos que crear una referencia al índice y luego asignar esta referencia a la transacción. El método **`index()`** crea esta referencia para el índice declarado en el atributo **`nombre`**.

deleteIndex(nombre) Si ya no necesitamos un índice podemos eliminarlo usando este método.

Transacciones

Un sistema de base de datos trabajando en un navegador debe contemplar algunas circunstancias únicas que no están presentes en otras plataformas. El navegador puede fallar, puede ser cerrado abruptamente, el proceso puede ser detenido por el usuario, o simplemente otro sitio web puede ser cargado en la misma ventana, por ejemplo. Existen muchas situaciones en las que trabajar directamente con la base de datos puede causar mal funcionamiento o incluso corrupción de datos. Para prevenir que algo así suceda, cada acción es realizada por medio de transacciones.

El método que genera una transacción se llama **`transaction()`.** Para declarar el tipo de transacción, contamos con los siguientes tres atributos:

READ_ONLY Este atributo genera una transacción de solo lectura. Modificaciones no son permitidas.

READ_WRITE Usando este tipo de transacción podemos leer y escribir. Modificaciones son permitidas.

`VERSION_CHANGE` Este tipo de transacción es solo utilizada para actualizar la versión de la base de datos.

Las más comunes son las transacciones de lectura y escritura. Sin embargo, para prevenir un uso inadecuado, el tipo **READ_ONLY** (solo lectura) es configurado por defecto. Por este motivo, cuando solo necesitamos obtener información de la base de datos, lo único que debemos hacer es declarar el destino de la transacción (normalmente el nombre del Almacén de Objetos de donde vamos a leer esta información).

Métodos de Almacenes de Objetos

Para interactuar con Almacenes de Objetos, leer y almacenar información, la API provee varios métodos:

add(objeto) Este método recibe un par clave/valor o un objeto conteniendo varios pares clave/valor, y con los datos provistos genera un objeto que es agregado al Almacén de Objetos seleccionado. Si un objeto con el mismo valor de índice ya existe, el método **`add()`** retorna un error.

put(objeto) Este método es similar al anterior, excepto que si ya existe un objeto con el mismo valor de índice lo sobrescribe. Es útil para modificar un objeto ya almacenado en el Almacén de Objetos seleccionado.

get(clave) Podemos leer un objeto específico del Almacén de Objetos usando este método. El atributo **`clave`** es el valor del índice del objeto que queremos leer.

delete(clave) Para eliminar un objeto del Almacén de Objetos seleccionado solo tenemos que llamar a este método con el valor del índice del objeto a eliminar.

11.2 Implementando IndexedDB

¡Suficiente con la teoría! Es momento de crear nuestra primera base de datos y aplicar algunos de los métodos ya mencionados en este capítulo. Vamos a simular una aplicación para almacenar información sobre películas. Puede agregar a la base los datos que usted desee, pero para referencia, de aquí en adelante vamos a mencionar los siguientes:

id: tt0068646 **nombre:** El Padrino **fecha:** 1972

id: tt0086567 **nombre:** Juegos de Guerra **fecha:** 1983

id: tt0111161 **nombre:** Cadena Perpetua **fecha:** 1994

id: tt1285016 **nombre:** La Red Social **fecha:** 2010

IMPORTANTE: Los nombres de las propiedades (`id`, `nombre` y `fecha`) son los que vamos a utilizar para nuestros ejemplos en el resto del capítulo. La información fue recolectada del sitio web www.imdb.com, pero usted puede utilizar su propia lista o información al azar para probar los códigos.

Plantilla

Como siempre, necesitamos un documento HTML y algunos estilos CSS para crear las cajas en pantalla que contendrán el formulario apropiado y la información retornada. El formulario nos permitirá insertar nuevas películas dentro de la base de datos solicitándonos una clave, el título y el año en el que la película fue realizada.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>IndexedDB API</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Clave:<br><input type="text" name="clave" id="clave"></p>
      <p>Título:<br><input type="text" name="texto" id="texto"></p>
      <p>Año:<br><input type="text" name="fecha" id="fecha"></p>
      <p><input type="button" name="grabar" id="grabar"
                                        value="Grabar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 11-1. *Plantilla para IndexedDB API.*

Los estilos CSS definen las cajas para el formulario y cómo mostrar la información en pantalla:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
```

```
  border: 1px solid #999999;
}
#clave, #texto{
  width: 200px;
}
#cajadatos > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

Listado 11-2. *Estilos para las cajas.*

Hágalo usted mismo: Necesitará un archivo HTML para la plantilla del Listado 11-1, un archivo CSS llamado **indexed.css** para los estilos del Listado 11-2 y un archivo Javascript llamado **indexed.js** para introducir todos los códigos estudiados a continuación.

Abriendo la base de datos

Lo primero que debemos hacer en el código Javascript es abrir la base de datos. El atributo **indexedDB** y el método **open()** abren la base con el nombre declarado o crean una nueva si no existe:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', agregarobjeto, false);

  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }

  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
```

Listado 11-3. *Abriendo la base de datos.*

La función **iniciar()** del Listado 11-3 prepara los elementos de la plantilla y abre la base de datos. La instrucción **indexedDB.open()** intenta abrir la base de datos con el nombre **mibase** y retorna el objeto **solicitud** con el resultado de la operación. Los eventos **error** o **success** son disparados sobre este objeto en caso de error o éxito, respectivamente.

IMPORTANTE: Al momento de escribir estas líneas la API se encuentra en estado experimental. Algunos atributos, incluyendo **indexedDB**, necesitan el prefijo del navegador para trabajar apropiadamente. Antes de abrir la base de datos en la función **iniciar()** detectamos la existencia de **webkitIndexedDB** o **mozIndexedDB** y preparamos los atributos para cada uno de estos navegadores específicos (Google Chrome o Firefox). Luego de que este período experimental termine podremos eliminar el condicional **if** al comienzo del código del Listado 11-3 y utilizar los métodos reales.

Los eventos son una parte importante de esta API. IndexedDB es una API síncrona y asíncrona. La parte síncrona está siendo desarrollada en estos momentos y está destinada a trabajar con la API Web Workers. En cambio, la parte asíncrona está destinada a un uso web normal y ya se encuentra disponible. Un sistema asíncrono realiza tareas detrás de escena y retorna los resultados posteriormente. Con este propósito, esta API dispara diferentes eventos en cada operación. Cada acción sobre la base de datos y su contenido es procesada detrás de escena (mientras el sistema ejecuta otros códigos) y los eventos correspondientes son disparados luego para informar los resultados obtenidos.

Luego de que la API procesa la solicitud para la base de datos, un evento **error** o **success** es disparado de

acuerdo al resultado y una de las funciones **`errores()`** o **`crear()`** es ejecutada para controlar los errores o continuar con la definición de la base de datos, respectivamente.

Versión de la base de datos

Antes de comenzar a trabajar en el contenido de la base de datos, debemos seguir algunos pasos para completar su definición. Como dijimos anteriormente, las bases de datos IndexedDB usan versiones. Cuando la base de datos es creada, un valor **`null`** (nulo) es asignado a su versión. Por lo tanto, controlando este valor podremos saber si la base de datos es nueva o no:

```
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }
}
```

Listado 11-4. Declarando la versión y respondiendo a eventos.

Nuestra función **`errores()`** es simple (no necesitamos procesar errores para esta aplicación de muestra). En este ejemplo solo usamos los atributos **`code`** y **`message`** de la interface IDBErrorEvent para generar un mensaje de alerta en caso de error. La función **`crear()`**, por otro lado, sigue los pasos correctos para detectar la versión de la base de datos y proveer un valor de versión en caso de que sea la primera vez que la aplicación es ejecutada en este ordenador. La función asigna el objeto **`result`** creado por el evento a la variable **`bd`** y usa esta variable para representar la base de datos (esta variable se definió como global para referenciar la base de datos en el resto del código).

> **IMPORTANTE:** En este momento algunos navegadores envían el objeto **`result`** a través del evento y otros a través del elemento que disparó el evento. Para seleccionar la referencia correcta de forma automática, usamos la lógica **`e.result || e.target.result`**. Seguramente usted deberá usar solo uno de estos valores en sus aplicaciones cuando las implementaciones estén listas.

La interface IDBDatabase provee la propiedad **`version`** para informar el valor de la versión actual y también provee el método **`setVersion()`** para declarar una nueva versión. Lo que hacemos en la función **`crear()`** en el Listado 11-4 es detectar el valor actual de la versión de la base de datos y declarar uno nuevo si es necesario (en caso de que el valor sea una cadena de texto vacía). Si la base de datos ya existe, el valor de la propiedad **`version`** será diferente de **`null`** (nulo) y no tendremos que configurar nada, pero si esta es la primera vez que el usuario utiliza esta aplicación entonces deberemos declarar un nuevo valor para la versión y configurar la base de datos.

El método **`setVersion()`** recibe una cadena de texto que puede ser un número o cualquier valor que se nos ocurra, solo debemos estar seguros de que siempre usamos el mismo valor en cada código para abrir la versión correcta de la base de datos. Este método es, así como cualquier otro procedimiento en esta API, asíncrono. La versión será establecida detrás de escena y el resultado será informado al código principal a través de eventos. Si ocurre un error en el proceso, llamamos a la función **`errores()`** como lo hicimos anteriormente, pero si la versión es establecida correctamente entonces la función **`crearbd()`** es llamada para declarar los Almacenes de Objetos e índices que usaremos en esta nueva versión.

Almacenes de Objetos e índices

A este punto debemos comenzar a pensar sobre la clase de objetos que vamos a almacenar y cómo vamos a leer más adelante la información contenida en los Almacenes de Objetos. Si hacemos algo mal o queremos agregar algo en la configuración de nuestra base de datos en el futuro deberemos declarar una nueva versión y migrar los datos desde la anterior, por lo que es importante tratar de organizar todo lo mejor posible desde el principio. Esto es debido a que la creación de Almacenes de Objetos e índices solo es posible durante una transacción **`setVersion`**.

```
function crearbd(){
  var almacen=bd.createObjectStore('peliculas',{keyPath:'id'});
  almacen.createIndex('BuscarFecha', 'fecha',{unique: false});
}
```

Listado 11-5. *Declarando Almacenes de Objetos e índices.*

Para nuestro ejemplo solo necesitamos un Almacén de Objetos (para almacenar películas) y dos índices. El primer índice, **`id`**, es declarado como el atributo **`clave`** para el método **`createObjectStore()`** cuando el Almacén de Objetos es creado. El segundo índice es asignado al Almacén de Objetos usando el método **`createIndex()`**. Este índice fue identificado con el nombre **`BuscarFecha`** y declarado para la propiedad **`fecha`** (esta propiedad es ahora un índice). Más adelante vamos a usar este índice para ordenar películas por año.

Agregando Objetos

Por el momento tenemos una base de datos con el nombre **`mibase`** que tendrá el valor de versión **`1.0`** y contendrá un Almacén de Objetos llamado **`peliculas`** con dos índices: **`id`** y **`fecha`**. Con esto ya podemos comenzar a agregar objetos en el almacén:

```
function agregarobjeto(){
  var clave=document.getElementById('clave').value;
  var titulo=document.getElementById('texto').value;
  var fecha=document.getElementById('fecha').value;

  var transaccion=bd.transaction(['peliculas'],
                                   IDBTransaction.READ_WRITE);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.add({id: clave, nombre: titulo, fecha:
                                                      fecha});
  solicitud.addEventListener('success', function(){
                                    mostrar(clave) }, false);

  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
  document.getElementById('fecha').value='';
}
```

Listado 11-6. *Agregando objetos.*

Al comienzo de la función **`iniciar()`** habíamos agregado al botón del formulario una escucha para el evento **`click`**. Esta escucha ejecuta la función **`agregarobjeto()`** cuando el evento es disparado (el botón es presionado). Esta función toma los valores de los campos del formulario (**`clave`**, **`texto`** y **`fecha`**) y luego genera una transacción para almacenar un nuevo objeto usando esta información.

Para comenzar la transacción, debemos usar el método **`transaction()`**, especificar el Almacén de Objetos involucrado en la transacción y el tipo de transacción a realizar. En este caso el almacén es **`peliculas`** y el tipo es declarado como **`READ_WRITE`**.

El próximo paso es seleccionar el Almacén de Objetos que vamos a usar. Debido a que la transacción puede ser originada para varios Almacenes de Objetos, tenemos que declarar cuál corresponde con la siguiente operación. Usando el método **`objectStore()`** abrimos el Almacén de Objetos y lo asignamos a la transacción con la siguiente línea: **`transaccion.objectStore('peliculas')`**.

Es momento de agregar el objeto al Almacén de Objetos. En este ejemplo usamos el método **`add()`** debido a que queremos crear un nuevo objeto, pero podríamos haber utilizado el método **`put()`** en su lugar si nuestra intensión hubiese sido modificar un viejo objeto. El método **`add()`** genera el objeto usando las propiedades **`id`**, **`nombre`** y **`fecha`** y las variables **`clave`**, **`titulo`** y **`fecha`**.

Finalmente, escuchamos al evento disparado por esta solicitud y ejecutamos la función **`mostrar()`** en caso de éxito. Existe también un evento **`error`**, por supuesto, pero como la respuesta a los errores depende de lo que usted quiera para su aplicación, no consideramos esa posibilidad en este ejemplo.

Leyendo Objetos

Si el objeto es correctamente almacenado, el evento **success** es disparado y la función **mostrar()** es ejecutada. En el código del Listado 11-6, esta función fue llamada dentro de una función anónima para poder pasar la variable **clave**. Ahora vamos a tomar este valor para leer el objeto previamente almacenado:

```
function mostrar(clave){
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.get(clave);
  solicitud.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var resultado=e.result || e.target.result;
  cajadatos.innerHTML='<div>'+resultado.id+' - '+resultado.nombre+'
                                    - '+resultado.fecha+'</div>';
}
```

Listado 11-7. *Leyendo y mostrando el objeto almacenado.*

El código del Listado 11-7 genera una transacción **READ_ONLY** y usa el método **get()** para leer el objeto con la clave recibida. No tenemos que declarar el tipo de transacción porque **READ_ONLY** es establecido por defecto.

El método **get()** retorna el objeto almacenado con la propiedad **id=clave**. Si, por ejemplo, insertamos la película *El Padrino* de nuestra lista, la variable **clave** tendrá el valor "tt0068646". Este valor es recibido por la función **mostrar()** y usado por el método **get()** para leer la película *El Padrino*. Como puede ver, este código es solo ilustrativo ya que solo puede retornar la misma película que acabamos de almacenar.

Como cada operación es asíncrona, necesitamos dos funciones para mostrar la información. La función **mostrar()** genera la transacción y lee el objeto, y la función **mostrarlista()** muestra los valores de las propiedades del objeto en pantalla. Otra vez, solo estamos escuchando al evento **success**, pero un evento **error** podría ser disparado en caso de que el objeto no pueda ser leído por alguna circunstancia en particular.

La función **mostrarlista()** recibe un objeto. Para acceder a sus propiedades solo tenemos que usar la variable representando al objeto y el nombre de la propiedad, como en **resultado.id** (la variable **resultado** representa el objeto e **id** es una de sus propiedades).

Finalizando el código

Del mismo modo que cualquier código previo, el ejemplo debe ser finalizado agregando una escucha para el evento **load** que ejecute la función **iniciar()** tan pronto como la aplicación es cargada en la ventana del navegador:

```
window.addEventListener('load', iniciar, false);
```

Listado 11-8. *Iniciando la aplicación.*

Hágalo usted mismo: Es momento de probar la aplicación en el navegador. Copie todos los códigos Javascript desde el Listado 11-3 al Listado 11-8 en el archivo **indexed.js** y abra el documento HTML del Listado 11-1. Usando el formulario en la pantalla, inserte información acerca de las películas listadas al comienzo de este capítulo. Cada vez que una nueva película es insertada, la misma información es mostrada en la caja de la derecha.

11.3 Listando datos

El método **`get()`** implementado en el código del Listado 11-7 solo retorna un objeto por vez (el último insertado). En el siguiente ejemplo vamos a usar un cursor para generar una lista incluyendo todas las películas almacenadas en el Almacén de Objetos **`peliculas`**.

Cursores

Los cursores son una alternativa ofrecida por la API para obtener y navegar a través de un grupo de objetos encontrados en la base de datos. Un cursor obtiene una lista específica de objetos de un Almacén de Objetos e inicia un puntero que apunta a un objeto de la lista a la vez.

Para generar un cursor, la API provee el método **`openCursor()`**. Este método extrae información del Almacén de Objetos seleccionado y retorna un objeto IDBCursor que tiene sus propios atributos y métodos para manipular el cursor:

continue() Este método mueve el puntero del cursor una posición y el evento **`success`** del cursor es disparado nuevamente. Cuando el puntero alcanza el final de la lista, el evento **`success`** es también disparado, pero retorna un objeto vacío. El puntero puede ser movido a una posición específica declarando un valor de índice dentro de los paréntesis.

delete() Este método elimina el objeto en la posición actual del cursor.

update(valor) Este método es similar a **`put()`** pero modifica el valor del objeto en la posición actual del cursor.

El método **`openCursor()`** también tiene atributos para especificar el tipo de objetos retornados y su orden. Los valores por defecto retornan todos los objetos disponibles en el Almacén de Objetos seleccionado, organizados en orden ascendente. Estudiaremos este tema más adelante.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('grabar');
  boton.addEventListener('click', agregarobjeto, false);
  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }
  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }else{
    mostrar();
  }
}
function crearbd(){
  var almacen=bd.createObjectStore('peliculas',{keyPath: 'id'});
  almacen.createIndex('BuscarFecha', 'fecha',{unique: false});
}
```

```
function agregarobjeto(){
  var clave=document.getElementById('clave').value;
  var titulo=document.getElementById('texto').value;
  var fecha=document.getElementById('fecha').value;
  var transaccion=bd.transaction(['peliculas'],
                                   IDBTransaction.READ_WRITE);
  var almacen=transaccion.objectStore('peliculas');
  var solicitud=almacen.add({id: clave, nombre: titulo, fecha:
                                                     fecha});
  solicitud.addEventListener('success', mostrar, false);
  document.getElementById('clave').value='';
  document.getElementById('texto').value='';
  document.getElementById('fecha').value='';
}
function mostrar(){
  cajadatos.innerHTML='';
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var cursor=almacen.openCursor();
  cursor.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
        '+cursor.value.nombre+' - '+cursor.value.fecha+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 11-9. *Lista de objetos.*

El Listado 11-9 muestra el código Javascript completo que necesitamos para este ejemplo. De las funciones usadas para configurar la base de datos, solo **crear()** presenta un pequeño cambio. Ahora, cuando la versión de la base de datos sea diferente a **null** (lo que significa que la base de datos ya ha sido creada) la función **mostrar()** es ejecutada. Esta función ahora se encuentra a cargo de mostrar la lista de objetos almacenados en el Almacén de Objetos, por lo que si la base de datos ya existe veremos una lista de objetos en la caja derecha de la pantalla tan pronto como la página web es cargada.

La mejora introducida en este código se encuentra en las funciones **mostrar()** y **mostrarlista()**. Aquí es donde trabajamos con cursores por primera vez.

Leer información de la base de datos con un cursor es también una operación que debe hacerse a través de una transacción. Por este motivo, lo primero que hacemos en la función **mostrar()** es generar una transacción del tipo **READ_ONLY** sobre el Almacén de Objetos **peliculas**. Este Almacén de Objetos es seleccionado como el involucrado en la transacción y luego el cursor es abierto sobre este almacén usando el método **openCursor()**.

Si la operación es exitosa, un objeto es retornado con toda la información obtenida del Almacén de Objetos, un evento **success** es disparado desde este objeto y la función **mostrarlista()** es ejecutada.

Para leer la información, el objeto retornado por la operación ofrece varios atributos:

key Este atributo retorna el valor de la clave del objeto en la posición actual del cursor.

value Este atributo retorna el valor de cualquier propiedad del objeto en la posición actual del cursor. El nombre de la propiedad debe ser especificado como una propiedad del atributo (por ejemplo, **value.fecha**).

direction Los objetos pueden ser leídos en orden ascendente o descendente (como veremos más adelante); este atributo retorna la condición actual en la cual son leídos.

count Este atributo retorna en número aproximado de objetos en el cursor.

En la función **mostrarlista()** del Listado 11-9, usamos el condicional **if** para controlar el contenido del cursor. Si ningún objeto es retornado o el puntero alcanza el final de la lista, entonces el objeto estará vacío y el bucle no es continuado. Sin embargo, cuando el puntero apunta a un objeto válido, la información es mostrada en pantalla y el puntero es movido hacia la siguiente posición con **continue()**.

Es importante mencionar que no debemos usar un bucle **while** aquí debido a que el método **continue()**

dispara nuevamente el evento **success** del cursor y la función completa es ejecutada para leer el siguiente objeto retornado.

> **Hágalo usted mismo:** El código del Listado 11-9 reemplaza todos los códigos Javascript previos. Vacíe el archivo **indexed.js** y copie en su interior este nuevo código. Abra la plantilla del Listado 11-1 y, si aún no lo hizo, inserte todas las películas del listado encontrado al comienzo de este capítulo. Verá la lista completa de películas en la caja derecha de la pantalla en orden ascendente, de acuerdo al valor de la propiedad **id**.

Cambio de orden

Hay dos detalles que necesitamos modificar para obtener la lista que queremos. Todas las películas en nuestro ejemplo están listadas en orden ascendente y la propiedad usada para organizar los objetos es **id**. Esta es la propiedad declarada como el atributo **clave** cuando el Almacén de Objetos **peliculas** fue creado, y es por tanto el índice usado por defecto. Pero ésta clase de valores no es lo que a nuestros usuarios normalmente les interesa.

Considerando esta situación, creamos otro índice en la función **crearbd()**. El nombre de este índice adicional fue declarado como **BuscarFecha** y la propiedad asignada al mismo es **fecha**. Este índice nos permitirá ordenar las películas de acuerdo al valor del año en el que fueron filmadas.

```
function mostrar(){
  cajadatos.innerHTML='';
  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var indice=almacen.index('BuscarFecha');

  var cursor=indice.openCursor(null, IDBCursor.PREV);
  cursor.addEventListener('success', mostrarlista, false);
}
```

Listado 11-10. *Orden descendiente por año.*

La función en el Listado 11-10 reemplaza a la función **mostrar()** del código del Listado 11-9. Esta nueva función genera una transacción, luego asigna el índice **BuscarFecha** al Almacén de Objetos usado en la transacción, y finalmente usa **openCursor()** para obtener los objetos que tienen la propiedad correspondiente al índice (en este caso **fecha**).

Existen dos atributos que podemos especificar en **openCursor()** para seleccionar y ordenar la información obtenida por el cursor. El primer atributo declara el rango dentro del cual los objetos serán seleccionados y el segundo es una de las siguientes constantes:

NEXT (siguiente). El orden de los objetos retornados será ascendente (este es el valor por defecto).

NEXT_NO_DUPLICATE (siguiente no duplicado). El orden de los objetos retornados será ascendente y los objetos duplicados serán ignorados (solo el primer objeto es retornado si varios comparten el mismo valor de índice).

PREV (anterior). El orden de los objetos retornados será descendente.

PREV_NO_DUPLICATE (anterior no duplicado). El orden de los objetos retornados será descendente y los objetos duplicados serán ignorados (solo el primer objeto es retornado si varios comparten el mismo valor de índice).

Con el método **openCursor()** usado en la función **mostrar()** en el Listado 11-10, obtenemos los objetos en orden descendente y declaramos el rango como **null**. Vamos a aprender cómo construir un rango y retornar solo los objetos deseados más adelante en este capítulo.

> **Hágalo usted mismo:** Tome el código anterior presentado en el Listado 11-9 y reemplace la función **show()** con la nueva función del Listado 11-10. Esta nueva función lista las películas en la pantalla por año y en orden descendente (las más nuevas primero). El resultado debería ser el siguiente:
>
> **id:** tt1285016 **nombre:** La Red Social **fecha:** 2010
>
> **id:** tt0111161 **nombre:** Cadena Perpetua **fecha:** 1994
>
> **id:** tt0086567 **nombre:** Juegos de Guerra **fecha:** 1983
>
> **id:** tt0068646 **nombre:** El Padrino **fecha:** 1972

11.4 Eliminando datos

Hemos aprendido cómo agregar, leer y listar datos. Es hora de estudiar la posibilidad de eliminar objetos de un Almacén de Objetos. Como mencionamos anteriormente, el método **`delete()`** provisto por la API recibe un valor y elimina el objeto con la clave correspondiente a ese valor.

El código es sencillo; solo necesitamos crear un botón para cada objeto listado en pantalla y generar una transacción **`READ_WRITE`** para poder realizar la operación y eliminar el objeto:

```
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
         '+cursor.value.nombre+' - '+cursor.value.fecha+' <button
             onclick="eliminar(\''+cursor.value.id+'\')">Eliminar
                                                    </button></div>';
    cursor.continue();
  }
}
function eliminar(clave){
  if(confirm('Está Seguro?')){
    var transaccion=bd.transaction(['peliculas'],
                                  IDBTransaction.READ_WRITE);
    var almacen=transaccion.objectStore('peliculas');
    var solicitud=almacen.delete(clave);
    solicitud.addEventListener('success', mostrar, false);
  }
}
```

Listado 11-11. *Eliminando objetos.*

El botón agregado a cada película en la función **`mostrarlista()`** del Listado 11-11 contiene un manejador de eventos en línea. Cada vez que el usuario hace clic en uno de estos botones, la función **`eliminar()`** es ejecutada con el valor de la propiedad **`id`** como su atributo. Esta función genera primero una transacción **`READ_WRITE`** y luego usando la clave recibida procede a eliminar el correspondiente objeto del Almacén de Objetos **`peliculas`**.

Al final, si la operación fue exitosa, el evento **`success`** es disparado y la función **`mostrar()`** es ejecutada para actualizar la lista de películas en pantalla.

> **Hágalo usted mismo:** Tome el código anterior del Listado 11-9, reemplace la función **`mostrarlista()`** y agregue la función **`eliminar()`** del código del Listado 11-11. Finalmente, abra el documento HTML del Listado 11-1 para probar la aplicación. Podrá ver el listado de películas pero ahora cada línea incluye un botón para eliminar la película del Almacén de Objetos. Experimente agregando y eliminando películas.

11.5 Buscando datos

Probablemente la operación más importante realizada en un sistema de base de datos es la búsqueda. El propósito absoluto de esta clase de sistemas es indexar la información almacenada para facilitar su posterior búsqueda. Como estudiamos anteriormente es este capítulo, el método **`get()`** es útil para obtener un objeto por vez cuando conocemos el valor de su clave, pero una operación de búsqueda es usualmente más compleja que esto.

Para obtener una lista específica de objetos desde el Almacén de Objetos, tenemos que pasar un rango como argumento para el método **`openCursor()`**. La API incluye la interface IDBKeyRange con varios métodos y propiedades para declarar un rango y limitar los objetos retornados:

only(valor) Solo los objetos con la clave que concuerda con **`valor`** son retornados. Por ejemplo, si buscamos películas por año usando **`only("1972")`**, solo la película *El Padrino* será retornada desde el almacén.

bound(bajo, alto, bajoAbierto, altoAbierto) Para realmente crear un rango, debemos contar con valores que indiquen el comienzo y el final de la lista, y además especificar si esos valores también serán incluidos. El valor del atributo **`bajo`** indica el punto inicial de la lista y el atributo **`alto`** es el punto final. Los atributos **`bajoAbierto`** y **`altoAbierto`** son valores booleanos usados para declarar si los objetos que concuerdan exactamente con los valores de los atributos **`bajo`** y **`alto`** serán ignorados. Por ejemplo, **`bound("1972", "2010", false, true)`** retornará una lista de películas filmadas desde el año 1972 hasta el año 2010, pero no incluirá las realizadas específicamente en el 2010 debido a que el valor es **`true`** (verdadero) para el punto donde el rango termina, lo que indica que el final es abierto y las películas de ese año no son incluidas.

lowerBound(valor, abierto) Este método crea un rango abierto que comenzará por **`valor`** e irá hacia arriba hasta el final de la lista. Por ejemplo, **`lowerBound("1983", true)`** retornará todas las películas hechas luego de 1983 (sin incluir las filmadas en ese año en particular).

upperBound(valor, abierto) Este es el opuesto al anterior. Creará un rango abierto, pero los objetos retornados serán todos los que posean un valor de índice menor a **`valor`**. Por ejemplo, **`upperBound("1983", false)`** retornará todas las películas hechas antes de 1983, incluyendo las realizadas ese mismo año (el atributo **`abierto`** fue declarado como **`false`**).

Preparemos primero una nueva plantilla para presentar un formulario en pantalla con el que se pueda buscar películas:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>IndexedDB API</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Buscar Película por Año:<br><input type="search"
                                   name="fecha" id="fecha"></p>
      <p><input type="button" name="buscar" id="buscar"
                                            value="Buscar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay información disponible
  </section>
</body>
</html>
```

Listado 11-12. *Formulario de búsqueda.*

Este nuevo documento HTML provee un botón y un campo de texto donde ingresar el año para buscar películas de acuerdo a un rango especificado en el siguiente código:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('buscar');
  boton.addEventListener('click', buscarobjetos, false);

  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }

  var solicitud=indexedDB.open('mibase');
  solicitud.addEventListener('error', errores, false);
  solicitud.addEventListener('success', crear, false);
}
function errores(e){
  alert('Error: '+e.code+' '+e.message);
}
function crear(e){
  bd=e.result || e.target.result;
  if(bd.version==''){
    var solicitud=bd.setVersion('1.0');
    solicitud.addEventListener('error', errores, false);
    solicitud.addEventListener('success', crearbd, false);
  }
}
function crearbd(){
  var almacen=bd.createObjectStore('peliculas', {keyPath: 'id'});
  almacen.createIndex('BuscarFecha', 'fecha', { unique: false });
}
function buscarobjetos(){
  cajadatos.innerHTML='';
  var buscar=document.getElementById('fecha').value;

  var transaccion=bd.transaction(['peliculas']);
  var almacen=transaccion.objectStore('peliculas');
  var indice=almacen.index('BuscarFecha');
  var rango=IDBKeyRange.only(buscar);

  var cursor=indice.openCursor(rango);
  cursor.addEventListener('success', mostrarlista, false);
}
function mostrarlista(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    cajadatos.innerHTML+='<div>'+cursor.value.id+' -
       '+cursor.value.nombre+' - '+cursor.value.fecha+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 11-13. *Buscando películas.*

La función **buscarobjetos()** es la más importante del Listado 11-13. En esta función generamos una transacción de solo lectura **READ_ONLY** para el Almacén de Objetos **peliculas**, seleccionamos el índice **BuscarFecha** para usar la propiedad **fecha** como índice, y creamos un rango desde el valor de la variable **buscar** (el año insertado en el formulario por el usuario). El método usado para construir el rango es **only()**, pero puede probar con cualquiera de los métodos estudiados. Este rango es pasado luego como un argumento del método **openCursor()**. Si la operación es exitosa, la función **mostrarlista()** imprimirá en pantalla la lista de películas del año seleccionado.

El método **only()** retorna solo las películas que concuerdan exactamente con el valor de la variable **buscar**.

Para probar otros métodos, puede proveer valores por defecto para completar el rango, por ejemplo **bound(buscar, “2011”, false, true)**.

El método **openCursor()** puede tomar dos posibles atributos al mismo tiempo. Por esta razón, una instrucción como **openCursor(rango, IDBCursor.PREV)** es válida y retornará los objetos en el rango especificado y en orden descendente (usando como referencia el mismo índice utilizado para el rango).

> **IMPORTANTE:** La característica de buscar textos se encuentra bajo consideración en este momento, pero aún no ha sido desarrollada o incluso incluida en la especificación oficial. Para obtener más información sobre esta API, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

11.6 Referencia rápida

La API IndexedDB tiene una infraestructura de bajo nivel. Los métodos y propiedades estudiados en este capítulo son solo parte de lo que esta API tiene para ofrecer. Con el propósito de simplificar los ejemplos, no seguimos ninguna estructura específica. Sin embargo, esta API, así como otras, está organizada en interfaces. Por ejemplo, existe una interface específica para tratar con la organización de la base de datos, otra para la creación y manipulación de Almacenes de Objetos, etc… Cada interface incluye sus propios métodos y propiedades, por lo que ahora vamos a presentar la información compartida en este capítulo siguiendo esta clasificación oficial.

> **IMPORTANTE:** Las descripciones presentadas en esta referencia rápida solo muestran los aspectos más relevantes de cada interface. Para estudiar la especificación completa, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Interface Environment (IDBEnvironment y IDBFactory)

La interface Environment, o IDBEnvironment, incluye un atributo IDBFactory. Juntas estas interfaces proveen los elementos necesarios para operar con bases de datos:

indexedDB Este atributo provee un mecanismo para acceder al sistema de base de datos indexado.

open(nombre) Este método abre una base de datos con el nombre especificado en su atributo. Si no existe una base de datos previa, una nueva es creada con el nombre provisto.

deleteDatabase(nombre) Este método elimina una base de datos con el nombre especificado en su atributo.

Interface Database (IDBDatabase)

El objeto retornado luego de la apertura o creación de una base de datos es procesado por esta interface. Con este objetivo, la interface provee varios métodos y propiedades:

version Esta propiedad retorna el valor de la versión actual de la base de datos abierta.

name Esta propiedad retorna el nombre de la base de datos abierta.

objectStoreNames Esta propiedad retorna un listado de los nombres de los Almacenes de Objetos contenidos dentro de la base de datos abierta.

setVersion(valor) Este método establece una nueva versión para la base de datos abierta. El atributo **`valor`** puede ser cualquier cadena de texto que deseemos.

createObjectStore(nombre, clave, autoIncremento) Este método crea un nuevo Almacén de Objetos para la base de datos abierta. El atributo **`nombre`** representa el nombre del Almacén de Objetos, **`clave`** es un índice común para todos los objetos almacenados en este almacén, y **`autoIncremento`** es un valor booleano que activa un generador automático de claves.

deleteObjectStore(nombre) Este método elimina un Almacén de Objetos con el nombre especificado en su atributo.

transaction(almacenes, tipo, máximo) Este método inicia una transacción con la base de datos. La transacción puede ser especificada para uno o más Almacenes de Objetos declarados en el atributo **`almacenes`**, y puede ser creada para diferentes tipos de acceso de acuerdo con el atributo **`tipo`**. Puede también recibir un atributo **`máximo`** en milisegundos para especificar el tiempo máximo permitido que la operación puede tardar en realizarse. Para mayor información sobre cómo configurar una transacción, ver Interface Transaction en esta Referencia rápida.

Interface Object Store (IDBObjectStore)

Esta interface incluye todos los métodos y propiedades necesarios para manipular objetos en un Almacén de Objetos (Object Store).

name Esta propiedad retorna el nombre del Almacén de Objetos actualmente en uso.

keyPath Esta propiedad retorna la **`clave`**, si existe, del Almacén de Objetos actualmente en uso (esta es la

clave definida como índice común en el momento en el que el Almacén de Objetos fue creado).

IndexNames Esta propiedad retorna una lista de los nombres de los índices creados para el Almacén de Objetos actualmente en uso.

add(objeto) Este método agrega un objeto al Almacén de Objetos seleccionado con la información provista en su atributo. Si un objeto con el mismo índice ya existe, un error es retornado. El método puede recibir un par clave/valor o un objeto conteniendo varios pares clave/valor como atributo.

put(objeto) Este método agrega un objeto al Almacén de Objetos seleccionado con la información provista en su atributo. Si un objeto con el mismo índice ya existe, el objeto es sobrescrito con la nueva información. El método puede recibir un par clave/valor o un objeto conteniendo varios pares clave/valor como atributo.

get(clave) Este método retorna el objeto con el valor del índice igual a `clave`.

delete(clave) Este método elimina el objeto con el valor del índice igual a `clave`.

createIndex(nombre, propiedad, único) Este método crea un nuevo índice para el Almacén de Objetos seleccionado. El atributo `nombre` especifica el nombre del índice, el atributo `propiedad` declara la propiedad de los objetos que será asociada con este índice, y el atributo `único` indica si los objetos con un mismo valor de índice serán permitidos o no.

index(nombre) Este método activa el índice con el nombre especificado en su atributo.

deleteIndex(nombre) Este método elimina el índice con el nombre especificado en su atributo.

openCursor(rango, dirección) Este método crea un cursor para el Almacén de Objetos seleccionado. El atributo `rango` es un objeto range (definido por la Interface Range) para determinar cuáles objetos son seleccionados. El atributo `dirección` establece el orden de estos objetos. Para mayor información sobre cómo configurar y manipular un cursor, ver la Interface Cursors en esta Referencia Rápida. Para mayor información sobre cómo construir un rango con el objeto range, ver la Interface Range en esta Referencia Rápida.

Interface Cursors (IDBCursor)

Esta interface provee valores de configuración para especificar el orden de los objetos seleccionados desde el Almacén de Objetos. Estos valores deben ser declarados como el segundo atributo del método `openCursor()`, como en `openCursor(null, IDBCursor.PREV)`.

NEXT (siguiente). Esta constante determina un orden ascendente para los objetos apuntados por el cursor (este es el valor por defecto).

NEXT_NO_DUPLICATE (siguiente no duplicado). Esta constante determina un orden ascendente para los objetos apuntados por el cursor e ignora los duplicados.

PREV (anterior). Esta constante determina un orden descendente para los objetos apuntados por el cursor.

PREV_NO_DUPLICATE (anterior no duplicado). Esta constante determina un orden descendente para los objetos apuntados por el cursor e ignora los duplicados.

Esta interface también provee varios métodos y propiedades para manipular los objetos apuntados por el cursor.

continue(clave) Este método mueve el puntero del cursor hacia el siguiente objeto en la lista o hacia el objeto referenciado por el atributo `clave`, si es declarado.

delete() Este método elimina el objeto que actualmente se encuentra apuntado por el cursor.

update(valor) Este método modifica el objeto actualmente apuntado por el cursor con el valor provisto por su atributo.

key Esta propiedad retorna el valor del índice del objeto actualmente apuntado por el cursor.

value Esta propiedad retorna el valor de cualquier propiedad del objeto actualmente apuntado por el cursor.

direction Esta propiedad retorna el orden de los objetos obtenidos por el cursor (ascendente o descendente).

Interface Transactions (IDBTransaction)

Esta interface provee valores de configuración para especificar el tipo de transacción que se va a llevar a cabo. Estos valores deben ser declarados como el segundo atributo del método **`transaction()`**, como en **`transaction(almacenes, IDBTransaction.READ_WRITE)`**.

***READ*_ONLY** Esta constante configura la transacción como una transacción de solo lectura (este es el valor por defecto).

READ_WRITE Esta constante configura la transacción como una transacción de lectura-escritura.

VERSION_CHANGE Este tipo de transacción es usado solamente para actualizar el número de versión de la base de datos.

Interface Range (IDBKeyRangeConstructors)

Esta interface provee varios métodos para la construcción de un rango a ser usado con cursores:

only(valor) Este método retorna un rango con los puntos de inicio y final iguales a **`valor`**.

bound(bajo, alto, bajoAbierto, altoAbierto) Este método retorna un rango con el punto de inicio declarado por **`bajo`**, el punto final declarado por **`alto`**, y si estos valores serán excluidos de la lista de objetos o no declarado por los últimos dos atributos.

lowerBound(valor, abierto) Este método retorna un rango comenzando por **`valor`** y terminando al final de la lista de objetos. El atributo **`abierto`** determina si los objetos que concuerdan con **`valor`** serán excluidos o no.

upperBound(valor, abierto) Este método retorna un rango comenzando desde el inicio de la lista de objetos y terminando en **`valor`**. El atributo **`abierto`** determina si los objetos que concuerdan con **`valor`** serán excluidos o no.

Interface Error (IDBDatabaseException)

Los errores retornados por las operaciones en la base de datos son informados a través de esta interface.

code Esta propiedad representa el número de error.

message Esta propiedad retorna un mensaje describiendo el error.

El valor retornado también puede ser comparado con las constantes de la siguiente lista para encontrar el error correspondiente.

UNKNOWN_ERR - valor 0.

NON_TRANSIENT_ERR - valor 1.

NOT_FOUND_ERR - valor 2.

CONSTRAINT_ERR - valor 3.

DATA_ERR - valor 4.

NOT_ALLOWED_ERR - valor 5.

TRANSACTION_INACTIVE_ERR - valor 6.

ABORT_ERR - valor 7.

READ_ONLY_ERR - valor 11.

RECOVERABLE_ERR - valor 21.

TRANSIENT_ERR - valor 31.

TIMEOUT_ERR - valor 32.

DEADLOCK_ERR - valor 33.

Capítulo 12
API File

12.1 Almacenamiento de archivos

Los archivos son unidades de información que usuarios pueden fácilmente compartir con otras personas. Los usuarios no pueden compartir el valor de una variable o un par clave/valor como los usados por la API Web Storage, pero ciertamente pueden hacer copias de sus archivos y enviarlos por medio de un DVD, memorias portátiles, discos duros, transmitirlos a través de Internet, etc… Los archivos pueden almacenar grandes cantidades de datos y ser movidos, duplicados o transmitidos independientemente de la naturaleza de su contenido.

Los archivos fueron siempre una parte esencial de cada aplicación, pero hasta ahora no había forma posible de trabajar con ellos en la web. Las opciones estaban limitadas a subir o descargar archivos ya existentes en servidores u ordenadores de usuarios. No existía la posibilidad de crear archivos, copiarlos o procesarlos en la web… hasta hoy.

La especificación de HTML5 fue desarrollada considerando cada aspecto necesario para la construcción y funcionalidad de aplicaciones web. Desde el diseño hasta la estructura elemental de los datos, todo fue cubierto. Y los archivos no podían ser ignorados, por supuesto. Por esta razón, la especificación incorpora la API File.

LA API File comparte algunas características con las API de almacenamiento estudiadas en capítulos previos. Esta API posee una infraestructura de bajo nivel, aunque no tan compleja como IndexedDB, y al igual que otras puede trabajar de forma síncrona o asíncrona. La parte síncrona fue desarrollada para ser usada con la API Web Workers (del mismo modo que IndexedDB y otras APIs), y la parte asíncrona está destinada a aplicaciones web convencionales. Estas características nos obligan nuevamente a cuidar cada aspecto del proceso, controlar si la operación fue exitosa o devolvió errores, y probablemente adoptar (o desarrollar nosotros mismos) en el futuro APIs más simples construidas sobre la misma.

API File es una vieja API que ha sido mejorada y expandida. Al día de hoy está compuesta por tres especificaciones: API File, API File: Directories & System, y API File: Writer, pero esta situación puede cambiar durante los siguientes meses con la incorporación de nuevas especificaciones o incluso la unificación de algunas de las ya existentes.

Básicamente, la API File nos permite interactuar con archivos locales y procesar su contenido en nuestra aplicación, la extensión la API File: Directories & System provee las herramientas para trabajar con un pequeño sistema de archivos creado específicamente para cada aplicación, y la extensión API File: Writer es para escribir contenido dentro de archivos que fueron creados o descargados por la aplicación.

12.2 Procesando archivos de usuario

Trabajar con archivos locales desde aplicaciones web puede ser peligroso. Los navegadores deben considerar medidas de seguridad antes de siquiera contemplar la posibilidad de dejar que las aplicaciones tengan acceso a los archivos del usuario. A este respecto, File API provee solo dos métodos para cargar archivos desde una aplicación: la etiqueta **`<input>`** y la operación arrastrar y soltar.

En el Capítulo 8 aprendimos cómo usar la API Drag and Drop para arrastrar archivos desde una aplicación de escritorio y soltarlos dentro de un espacio en la página web. La etiqueta **`<input>`** (también estudiada en capítulos anteriores), cuando es usada con el tipo **`file`**, trabaja de forma similar a API Drag and Drop. Ambas técnicas transmiten archivos a través de la propiedad **`files`**. Del mismo modo que lo hicimos en ejemplos previos, lo único que debemos hacer es explorar el valor de esta propiedad para obtener cada archivo que fue seleccionado o arrastrado.

> **IMPORTANTE:** Esta API y sus extensiones no trabajan en este momento desde un servidor local, y solo Google Chrome y Firefox tienen implementaciones disponibles. Algunas de estas implementaciones son tan nuevas que solo trabajan en navegadores experimentales como Chromium (*www.chromium.org*) o Firefox Beta. Para ejecutar los códigos de este capítulo, deberá usar las últimas versiones de navegadores disponibles y subir todos los archivos a su servidor.

Plantilla

En esta primera parte del capítulo vamos a usar la etiqueta **`<input>`** para seleccionar archivos, pero usted puede, si lo desea, aprovechar la información en el Capítulo 8 para integrar estos códigos con API Drag and Drop.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>File API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Archivos:<br><input type="file" name="archivos"
                                        id="archivos"></p>
    </form>
  </section>
  <section id="cajadatos">
    No se seleccionaron archivos
  </section>
</body>
</html>
```

Listado 12-1. *Plantilla para trabajar con los archivos del usuario.*

El archivo CSS incluye estilos para esta plantilla y otros que vamos a usar más adelante:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

```
.directorio{
  color: #0000FF;
  font-weight: bold;
  cursor: pointer;
}
```

Listado 12-2. *Estilos para el formulario y la* `cajadatos`.

Leyendo archivos

Para leer archivos en el ordenador de los usuarios tenemos que usar la interface **`FileReader`**. Esta interface retorna un objeto con varios métodos para obtener el contenido de cada archivo:

readAsText(archivo, codificación) Para procesar el contenido como texto podemos usar este método. Un evento **`load`** es disparado desde el objeto **`FileReader`** cuando el archivo es cargado. El contenido es retornado codificado como texto UTF-8 a menos que el atributo **`codificación`** sea especificado con un valor diferente. Este método intentará interpretar cada byte o una secuencia de múltiples bytes como caracteres de texto.

readAsBinaryString(archivo) La información es leída por este método como una sucesión de enteros en el rango de 0 a 255. Este método nos asegura que cada byte es leído como es, sin ningún intento de interpretación. Es útil para procesar contenido binario como imágenes o videos.

readAsDataURL(archivo) Este método genera una cadena del tipo data:url codificada en base64 que representa los datos del archivo.

readAsArrayBuffer(archivo) Este método retorna los datos del archivo como datos del tipo ArrayBuffer.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', procesar, false);
}
function procesar(e){
  var archivos=e.target.files;
  var archivo=archivos[0];
  var lector=new FileReader();
  lector.onload=mostrar;
  lector.readAsText(archivo);
}
function mostrar(e){
  var resultado=e.target.result;
  cajadatos.innerHTML=resultado;
}
window.addEventListener('load', iniciar, false);
```

Listado 12-3. *Leyendo un archivo de texto.*

Desde el campo **`archivos`** del documento HTML del Listado 12-1 el usuario puede seleccionar un archivo para ser procesado. Para detectar esta acción, en la función **`iniciar()`** del Listado 12-3 agregamos una escucha para el evento **`change`**. De este modo, la función **`procesar()`** será ejecutada cada vez que algo cambie en el elemento **`<input>`** (un nuevo archivo es seleccionado).

La propiedad **`files`** enviada por el elemento **`<input>`** (y también por la API Drag and Drop) es un array conteniendo todos los archivos seleccionados. Cuando el atributo **`multiple`** no está presente en la etiqueta **`<input>`** no es posible seleccionar múltiples archivos, por lo que el primer elemento del array será el único disponible. Al comienzo de la función **`procesar()`** tomamos el contenido de la propiedad **`files`**, lo asignamos a la variable **`archivos`** y luego seleccionamos el primer archivo con la línea **`var archivo=archivos[0]`**.

IMPORTANTE: Para aprender más acerca del atributo **`multiple`** lea nuevamente el Capítulo 6, Listado 6-17. También puede encontrar un ejemplo de cómo trabajar con múltiples archivos en el código del Listado 8-10, Capítulo 8.

Lo primero que debemos hacer para comenzar a procesar el archivo es obtener un objeto **`FileReader`**

usando el constructor **`FileReader()`**. En la función **`procesar()`** del Listado 12-3 llamamos a este objeto **`lector`**. En el siguiente paso, registramos un manejador de eventos **`onload`** para el objeto **`lector`** con el objetivo de detectar cuando el archivo fue cargado y ya podemos comenzar a procesarlo. Finalmente, el método **`readAsText()`** lee el archivo y retorna su contenido en formato texto.

Cuando el método **`readAsText()`** finaliza la lectura del archivo, el evento **`load`** es disparado y la función **`mostrar()`** es llamada. Esta función toma el contenido del archivo desde la propiedad **`result`** del objeto **`lector`** y lo muestra en pantalla.

Este código, por supuesto, espera recibir archivos de texto, pero el método **`readAsText()`** toma lo que le enviamos y lo interpreta como texto, incluyendo archivos con contenido binario (por ejemplo, imágenes). Si carga archivos con diferente contenido, verá caracteres extraños aparecer en pantalla.

> **Hágalo usted mismo:** Cree archivos con los códigos de los Listados 12-1, 12-2 y 12-3. Los nombres para los archivos CSS y Javascript fueron declarados en el documento HTML como **`file.css`** y **`file.js`** respectivamente. Abra la plantilla en el navegador y use el formulario para seleccionar un archivo en su ordenador. Intente con archivos de texto así como imágenes para ver cómo los métodos utilizados presentan el contenido en pantalla.

> **IMPORTANTE:** En este momento, API File y cada una de sus especificaciones están siendo implementadas por los fabricantes de navegadores. Los códigos en esta capítulo fueron testeados en Google Chrome y Firefox 4+, pero las últimas versiones de Chrome no habían implementado aún el método **`addEventListener()`** para **`FileReader`** y otros objetos. Por esta razón, usamos manejadores de eventos en nuestro ejemplo, como **`onload`**, cada vez que era necesario para que el código trabajara correctamente. Por ejemplo, **`lector.onload=mostrar`** fue usado en lugar de **`lector.addEventListener('load', mostrar, false)`**. Como siempre, le recomendamos probar los códigos en cada navegador disponible para encontrar qué implementación trabaja correctamente con esta API.

Propiedades de archivos

En una aplicación real, información como el nombre del archivo, su tamaño o tipo será necesaria para informar al usuario sobre los archivos que están siendo procesados o incluso controlar cuáles son o no son admitidos. El objeto enviado por el elemento **`<input>`** incluye varias propiedades para acceder a esta información:

> **name** Esta propiedad retorna el nombre completo del archivo (nombre y extensión).
>
> **size** Esta propiedad retorna el tamaño del archivo en bytes.
>
> **type** Esta propiedad retorna el tipo de archivo, especificado en tipos MIME.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', procesar, false);
}
function procesar(e){
  var archivos=e.target.files;
  cajadatos.innerHTML='';
  var archivo=archivos[0];
  if(!archivo.type.match(/image.*/i)){
    alert('seleccione una imagen');
  }else{
    cajadatos.innerHTML+='Nombre: '+archivo.name+'<br>';
    cajadatos.innerHTML+='Tamaño: '+archivo.size+' bytes<br>';

    var lector=new FileReader();
    lector.onload=mostrar;
    lector.readAsDataURL(archivo);
  }
}
function mostrar(e){
  var resultado=e.target.result;
  cajadatos.innerHTML+='<img src="'+resultado+'">';
}
window.addEventListener('load', iniciar, false);
```

Listado 12-4. Cargando imágenes.

El ejemplo del Listado 12-4 es similar al anterior excepto que esta vez usamos el método **readAsDataURL()** para leer el archivo. Este método retorna el contenido del archivo en el formato data:url que puede ser usado luego como fuente de un elemento **<img>** para mostrar la imagen seleccionada en la pantalla.

Cuando necesitamos procesar una clase particular de archivo, lo primero que debemos hacer es leer la propiedad **type** del archivo. En la función **procesar()** del Listado 12-4 controlamos este valor aprovechando el viejo método **match()**. Si el archivo no es una imagen, mostramos un mensaje de error con **alert()**. Si, por otro lado, el archivo es efectivamente una imagen, el nombre y tamaño del archivo son mostrados en pantalla y el archivo es abierto.

A pesar del uso de **readAsDataURL()**, el proceso de apertura es exactamente el mismo. El objeto **FileReader** es creado, el manejador **onload** es registrado y el archivo es cargado. Una vez que el proceso termina, la función **mostrar()** usa el contenido de la propiedad **result** como fuente del elemento **<img>** y la imagen seleccionada es mostrada en la pantalla.

Conceptos básicos: Para construir el filtro aprovechamos Expresiones Regulares y el conocido método Javascript **match()**. Este método busca por cadenas de texto que concuerden con la expresión regular, retornando un array con todas las coincidencias o el valor **null** en caso de no encontrar ninguna. El tipo MIME para imágenes es algo como **image/jpeg** para imágenes en formato JPG, o **image/gif** para imágenes en formato GIF, por lo que la expresión **/image.*/i** aceptará cualquier formato de imagen, pero solo permitirá que imágenes y no otro tipo de archivos sean leídos. Para más información sobre Expresiones Regulares o tipos MIME, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Blobs

Además de archivos, la API trabaja con otra clase de fuente de datos llamada blobs. Un blob es un objeto representando datos en crudo. Fueron creados con el propósito de superar limitaciones de Javascript para trabajar con datos binarios. Un blob es normalmente generado a partir de un archivo, pero no necesariamente. Es una buena alternativa para trabajar con datos sin cargar archivos enteros en memoria, y provee la posibilidad de procesar información binaria en pequeñas piezas.

Un blob tiene propósitos múltiples, pero está enfocado en ofrecer una mejor manera de procesar grandes piezas de datos crudos o archivos. Para generar blobs desde otros blobs o archivos, la API ofrece el método **slice()**:

slice(comienzo, largo, tipo) Este método retorna un nuevo blob generado desde otro blob o un archivo. El primer atributo indica el punto de comienzo, el segundo el largo del nuevo blob, y el último es un atributo opcional para especificar el tipo de datos usados.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', procesar, false);
}
function procesar(e){
  var archivos=e.target.files;
  cajadatos.innerHTML='';
  var archivo=archivos[0];
  var lector=new FileReader();
  lector.onload=function(e){ mostrar(e, archivo); };
  var blob=archivo.slice(0,1000);
  lector.readAsBinaryString(blob);
}
function mostrar(e, archivo){
  var resultado=e.target.result;
  cajadatos.innerHTML='Nombre: '+archivo.name+'<br>';
  cajadatos.innerHTML+='Tipo: '+archivo.type+'<br>';
  cajadatos.innerHTML+='Tamaño: '+archivo.size+' bytes<br>';
  cajadatos.innerHTML+='Tamaño Blob: '+resultado.length+'
                                                bytes<br>';
  cajadatos.innerHTML+='Blob: '+resultado;
}
```

```
window.addEventListener('load', iniciar, false);
```

Listado 12-5. *Trabajando con blobs.*

IMPORTANTE: Debido a inconsistencias con métodos previos, un reemplazo para el método **`slice`** está siendo implementado en este momento. Hasta que este método esté disponible, para probar el código del Listado 12-5 en las últimas versiones de Firefox y Google Chrome tendrá que reemplazar **`slice`** por **`mozSlice`** y **`webkitSlice`** respectivamente. Para más información, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

En el código del Listado 12-5, hicimos exactamente lo que veníamos haciendo hasta el momento, pero esta vez en lugar de leer el archivo completo creamos un blob con el método **`slice()`**. El blob tiene 1000 bytes de largo y comienza desde el byte 0 del archivo. Si el archivo cargado es más pequeño que 1000 bytes, el blob será del mismo largo del archivo (desde el comienzo hasta el EOF, o End Of File).

Para mostrar la información obtenida por este proceso, registramos el manejador de eventos **`onload`** y llamamos a la función **`mostrar()`** desde una función anónima con la que pasamos la referencia del objeto **`archivo`**. Este objeto es recibido por **`mostrar()`** y sus propiedades son mostradas en la pantalla.

Las ventajas ofrecidas por blobs son incontables. Podemos crear un bucle para dividir un archivo en varios blobs, por ejemplo, y luego procesar esta información parte por parte, creando programas para subir archivos al servidor o aplicaciones para procesar imágenes, entre otras. Los blobs ofrecen nuevas posibilidades a los códigos Javascript.

Eventos

El tiempo que toma a un archivo para ser cargado en memoria depende de su tamaño. Para archivos pequeños, el proceso se asemeja a una operación instantánea, pero grandes archivos pueden tomar varios segundos en cargar. Además del evento **`load`** ya estudiado, la API provee eventos especiales para informar sobre cada instancia del proceso:

loadstart Este evento es disparado desde el objeto **`FileReader`** cuando la carga comienza.

progress Este evento es disparado periódicamente mientras el archivo o blob está siendo leído.

abort Este evento es disparado si el proceso es abortado.

error Este evento es disparado cuando la carga ha fallado.

loadend Este evento es similar a **`load`**, pero es disparado en caso de éxito o fracaso.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', procesar, false);
}
function procesar(e){
  var archivos=e.target.files;
  cajadatos.innerHTML='';
  var archivo=archivos[0];
  var lector=new FileReader();
  lector.onloadstart=comenzar;
  lector.onprogress=estado;
  lector.onloadend=function(){ mostrar(archivo); };
  lector.readAsBinaryString(archivo);
}
function comenzar(e){
  cajadatos.innerHTML='<progress value="0" max="100">0%</progress>';
}
function estado(e){
  var por=parseInt(e.loaded/e.total*100);
  cajadatos.innerHTML='<progress value="'+por+'" max="100">'
                                              +por+'%</progress>';
}
function mostrar(archivo){
  cajadatos.innerHTML='Nombre: '+archivo.name+'<br>';
```

```
  cajadatos.innerHTML+='Tipo: '+archivo.type+'<br>';
  cajadatos.innerHTML+='Tamaño: '+archivo.size+' bytes<br>';
}
window.addEventListener('load', iniciar, false);
```

Listado 12-6. *Usando eventos para controlar el proceso de lectura.*

Con el código del Listado 12-6 creamos una aplicación que carga un archivo y muestra el progreso de la operación a través de una barra de progreso. Tres manejadores de eventos fueron registrados en el objeto **`FileReader`** para controlar el proceso de lectura y dos funciones fueron creadas para responder a estos eventos: **`comenzar()`** y **`estado()`**. La función **`comenzar()`** iniciará la barra de progreso con el valor 0% y la mostrará en pantalla. Esta barra de progreso podría usar cualquier valor o rango, pero decidimos usar porcentajes para que sea más comprensible para el usuario. En la función **`estado()`**, este porcentaje es calculado a partir de las propiedades **`loaded`** y **`total`** retornadas por el evento **`progress`**. La barra de progreso es recreada en la pantalla cada vez que el evento **`progress`** es disparado.

> **Hágalo usted mismo:** Usando la plantilla del Listado 12-1 y el código Javascript del Listado 12-6, pruebe cargar un archivo extenso (puede intentar con un video, por ejemplo) para ver la barra de progreso en funcionamiento. Si el navegador no reconoce el elemento **`<progress>`**, el contenido de este elemento es mostrado en su lugar.

> **IMPORTANTE:** En nuestro ejemplo utilizamos **`innerHTML`** para agregar un nuevo elemento **`<progress>`** al documento. Esta no es una práctica recomendada pero es útil y conveniente por razones didácticas. Normalmente los elementos son agregados al documento usando el método Javascript **`createElement()`** junto con **`appendChild()`**.

12.3 Creando archivos

La parte principal de API File es útil para cargar y procesar archivos ubicados en el ordenador del usuario, pero toma archivos que ya existen en el disco duro. No contempla la posibilidad de crear nuevos archivos o directorios. Una extensión de esta API llamada API File: Directories & System se hace cargo de esta situación. La API reserva un espacio específico en el disco duro, un espacio de almacenamiento especial en el cual la aplicación web podrá crear y procesar archivos y directorios exactamente como una aplicación de escritorio lo haría. El espacio es único y solo accesible por la aplicación que lo creó.

IMPORTANTE: Al momento de escribir estas líneas Google Chrome es el único navegador que ha implementado esta extensión de API File, pero no reserva espacio de almacenamiento por defecto. Si intentamos ejecutar los siguientes códigos un error QUOTA_EXCEEDED (cupo excedido) será mostrado. Para poder usar API File: Directories & System, Chrome debe ser abierto con la siguiente bandera: **`--unlimited-quota-for-files`**. Para incorporar esta bandera en Windows, vaya a su escritorio, haga clic con el botón derecho del ratón sobre el ícono de Google Chrome y seleccione la opción Propiedades. Dentro de la ventana abierta verá un campo llamado Destino con la ruta y el nombre del archivo del programa. Al final de esta línea agregue la bandera **`--unlimited-quota-for-files`**. La ruta obtenida será similar a la siguiente:

C:\Usuarios\...\Chrome\Application\chrome.exe --unlimited-quota-for-files

Plantilla

Para probar esta parte de la API vamos a necesitar un nuevo formulario con un campo de texto y un botón para crear y procesar archivos y directorios:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>File API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Nombre:<br><input type="text" name="entrada"
                                     id="entrada" required></p>
      <p><input type="button" name="boton" id="boton"
                                          value="Aceptar"></p>
    </form>
  </section>
  <section id="cajadatos">
    No hay entradas disponibles
  </section>
</body>
</html>
```

Listado 12-7. *Nueva plantilla para File API: Directories & System.*

Hágalo Usted Mismo: El documento HTML genera un nuevo formulario pero preserva la misma estructura y estilos CSS. Solo necesita reemplazar el código HTML anterior por el del Listado 12-7 y copiar los códigos Javascript dentro del archivo **`file.js`** para probar los siguientes ejemplos.

IMPORTANTE: El atributo **`request`** fue incluido en el elemento **`<input>`**, pero no será considerado en los códigos de este capítulo. Para volver efectivo el proceso de validación, deberemos aplicar API Forms. Lea el código del Listado 10-5, Capítulo 10, para encontrar un ejemplo sobre cómo hacerlo.

El disco duro

El espacio reservado para la aplicación es como un espacio aislado, una pequeña unidad de disco duro con su propio directorio raíz y configuración. Para comenzar a trabajar con esta unidad virtual, primero tenemos que solicitar que un Sistema de Archivos sea inicializado para nuestra aplicación.

requestFileSystem(tipo, tamaño, función éxito, función error) Este método crea el Sistema de Archivos del tamaño y tipo especificados por sus atributos. El valor del atributo **`tipo`** puede ser **`TEMPORARY`** (temporario) o **`PERSISTENT`** (persistente) de acuerdo al tiempo que deseamos que los archivos sean preservados. El atributo **`tamaño`** determina el espacio total que será reservado en el disco duro para este Sistema de Archivos en bytes. En caso de error o éxito, el método llama a las correspondientes funciones.

El método **`requestFileSystem()`** retorna un objeto FileSystem con dos propiedades:

root El valor de esta propiedad es una referencia al directorio raíz del Sistema de Archivos. Este es también un objeto **`DirectoryEntry`** (Entrada de Directorio) y tiene los métodos asignados a esta clase de objetos, como veremos más adelante. Usando esta propiedad podemos referenciar el espacio de almacenamiento y por medio de esta referencia trabajar con archivos y directorios.

name Esta propiedad retorna información acerca del Sistema de Archivos, como el nombre asignado por el navegador y su condición.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', crear, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                              creardd, errores);
}
function creardd(sistema) {
  dd=sistema.root;
}
function crear(){
  var nombre=document.getElementById('entrada').value;
  if(nombre!=''){
    dd.getFile(nombre, {create: true, exclusive: false}, mostrar,
                                                       errores);
  }
}
function mostrar(entrada){
  document.getElementById('entrada').value='';

  cajadatos.innerHTML='Entrada Creada!<br>';
  cajadatos.innerHTML+='Nombre: '+entrada.name+'<br>';
  cajadatos.innerHTML+='Ruta: '+entrada.fullPath+'<br>';
  cajadatos.innerHTML+='Sistema: '+entrada.filesystem.name;
}
function errores(e){
  alert('Error: '+e.code);
}
window.addEventListener('load', iniciar, false);
```

Listado 12-8. *Creando nuestro propio Sistema de Archivos.*

IMPORTANTE: Google Chrome es el único navegador en este momento con una implementación funcional de esta parte de File API. Debido a que la implementación es experimental, tuvimos que reemplazar el método **`requestFileSystem()`** por el específico de Chrome **`webkitRequestFileSystem()`**. Usando este método, podrá probar en su navegador los códigos para éste y los siguientes ejemplos. Una vez que la etapa de experimentación esté finalizada podrá volver a usar el método original.

Usando el documento HTML del Listado 12-7 y el código del Listado 12-8, obtenemos nuestra primera aplicación para trabajar con nuevos archivos en el ordenador del usuario. El código llama al método **`requestFileSystem()`** para crear el Sistema de Archivos (u obtener una referencia si el sistema ya existe), y si esta es la primera visita, el Sistema de Archivos es creado como permanente, con un tamaño de 5 megabytes (5*1024*1024). En caso de que esta última operación sea un éxito, la función **`creardd()`** es ejecutada, continuando con el proceso de inicialización. Para controlar errores, usamos la función **`errores()`**, del mismo modo que lo hicimos para otras APIs.

Cuando el Sistema de Archivos es creado o abierto, la función **creardd()** recibe un objeto **FileSystem** y graba el valor de la propiedad **root** en la variable **dd** para referenciar el directorio raíz más adelante.

Creando archivos

El proceso de iniciación del Sistema de Archivos ha sido finalizado. El resto de las funciones en el código del Listado 12-8 crean un nuevo archivo y muestran los datos de la entrada (un archivo o directorio) en la pantalla. Cuando el botón "Aceptar" es presionado en el formulario, la función **crear()** es llamada. Esta función asigna el texto insertado en el elemento **<input>** a la variable **nombre** y crea un archivo con ese nombre usando el método **getFile()**.

Este último método es parte de la interface **DirectoryEntry** incluida en la API. La interface provee un total de cuatro métodos para crear y manejar archivos y directorios:

getFile(ruta, opciones, función éxito, función error) Este método crea o abre un archivo. El atributo **ruta** debe incluir el nombre del archivo y la ruta donde el archivo está localizado (desde la raíz de nuestro Sistema de Archivos). Hay dos banderas que podemos usar para configurar el comportamiento de este método: **create** y **exclusive**. Ambas reciben valores booleanos. La bandera **create** (crear) indica si el archivo será creado o no (en caso de que no exista, por supuesto), y la bandera **exclusive** (exclusivo), cuando es declarada como **true** (verdadero), fuerza al método **getFile()** a retornar un error si intentamos crear un archivo que ya existe. Este método también recibe dos funciones para responder en case de éxito o error.

getDirectory(ruta, opciones, función éxito, función error) Este método tiene exactamente las mismas características que el anterior pero es exclusivo para directorios (carpetas).

createReader() Este método retorna un objeto **DirectoryReader** para leer entradas desde un directorio específico.

removeRecursively() Este es un método específico para eliminar directorios y todo su contenido.

En el código del Listado 12-8, el método **getFile()** usa el valor de la variable **nombre** para crear u obtener el archivo. El archivo será creado si no existe (**create: true**) o será leído en caso contrario (**exclusive: false**). La función **crear()** también controla que el valor de la variable **nombre** no sea una cadena vacía antes de ejecutar **getFile()**.

El método **getFile()** usa dos funciones, **mostrar()** y **errores()**, para responder al éxito o fracaso de la operación. La función **mostrar()** recibe un objeto **Entry** (**entrada**) y muestra el valor de sus propiedades en la pantalla. Este tipo de objetos tiene varios métodos y propiedades asociadas que estudiaremos más adelante. Por ahora hemos aprovechado solo las propiedades **name**, **fullPath** y **filesystem**.

Creando directorios

El método **getFile()** (específico para archivos) y el método **getDirectory()** (específico para directorios) son exactamente iguales. Para crear un directorio (carpeta) en nuestro Sistema de Archivos del ejemplo anterior, solo tenemos que reemplazar el nombre **getFile()** por **getDirectory()**, como es mostrado en el siguiente código:

```
function crear(){
  var nombre=document.getElementById('entrada').value;
  if(nombre!=''){
    dd.getDirectory(nombre, {create: true, exclusive: false},
                                          mostrar, errores);
  }
}
```

Listado 12-9. *Usando `getDirectory()` para crear un directorio.*

Ambos métodos son parte del objeto **DirectoryEntry** llamado **root**, que estamos representando con la variable **dd**, por lo que siempre deberemos usar esta variable para llamar a los métodos y crear archivos y directorios en el Sistema de Archivos de nuestra aplicación.

Hágalo usted mismo: Use la función en el Listado 12-9 para reemplazar la función **crear()** del Listado 12-8 y así crear directorios en lugar de archivos. Suba los archivos a su servidor, abra el documento HTML del

Listado 12-7 en su navegador y cree un directorio usando el formulario en la pantalla.

Listando archivos

Como mencionamos antes, el método **`createReader()`** nos permite acceder a una lista de entradas (archivos y directorios) en una ruta específica. Este método retorna un objeto **`DirectoryReader`** que contiene el método **`readEntries()`** para leer las entradas obtenidas:

readEntries(función éxito, función error) Este método lee el siguiente bloque de entradas desde el directorio seleccionado. Cada vez que el método es llamado, la función utilizada para procesar operaciones exitosas retorna un objeto con la lista de entradas o el valor **`null`** si no se encontró ninguna.

El método **`readEntries()`** lee la lista de entradas por bloque. Como consecuencia, no existe garantía alguna de que todas las entradas serán retornadas en una sola llamada. Tendremos que llamar al método tantas veces como sea necesario hasta que el objeto retornado sea un objeto vacío.

Además, deberemos hacer otra consideración antes de escribir nuestro próximo código. El método **`createReader()`** retorna un objeto **`DirectoryReader`** para un directorio específico. Para obtener los archivos que queremos, primero tenemos que obtener el correspondiente objeto **`Entry`** del directorio que queremos leer usando el ya conocido método **`getDirectory()`**:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', crear, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                              creardd, errores);
}
function creardd(sistema) {
  dd=sistema.root;
  ruta='';
  mostrar();
}
function errores(e){
  alert('Error: '+e.code);
}
function crear(){
  var nombre=document.getElementById('entrada').value;
  if(nombre!=''){
    nombre=ruta+nombre;
    dd.getFile(nombre, {create: true, exclusive: false}, mostrar,
                                                       errores);
  }
}
function mostrar(){
  document.getElementById('entrada').value='';

  cajadatos.innerHTML='';
  dd.getDirectory(ruta,null,leerdir,errores);
}
function leerdir(dir){
  var lector=dir.createReader();
  var leer=function(){
      lector.readEntries(function(archivos){
          if(archivos.length){
            listar(archivos);
            leer();
          }
        }, errores);
    }
  leer();
}
```

```
function listar(archivos){
  for(var i=0; i<archivos.length; i++) {
    if(archivos[i].isFile) {
      cajadatos.innerHTML+=archivos[i].name+'<br>';
    }else if(archivos[i].isDirectory){
      cajadatos.innerHTML+='<span onclick=
                          "cambiardir(\''+archivos[i].name+'\')"
          class="directorio">+'+archivos[i].name+'</span><br>';
    }
  }
}
function cambiardir(nuevaruta){
  ruta=ruta+nuevaruta+'/';
  mostrar();
}
window.addEventListener('load', iniciar, false);
```

Listado 12-10. *Sistema de Archivos completo.*

Este código no reemplazará al Explorador de Archivos de Windows, pero al menos provee toda la información que necesitamos para entender cómo construir un Sistema de Archivos útil y funcional para la web. Vamos a analizarlo parte por parte.

La función **`iniciar()`** hace lo mismo que en códigos previos: inicia o crea el Sistema de Archivos y llama a la función **`creardd()`** si tiene éxito. Además de declarar la variable **`dd`** para referenciar el directorio raíz de nuestro disco duro virtual, la función **`creardd()`** también inicializa la variable **`ruta`** con una cadena de texto vacía (representando el directorio raíz) y llama a la función **`mostrar()`** para mostrar la lista de entradas en pantalla tan pronto como la aplicación es cargada.

La variable **`ruta`** será usada en el resto de la aplicación para conservar el valor de la ruta actual dentro de Sistema de Archivos en la que el usuario está trabajando. Para entender su importancia, puede ver cómo la función **`crear()`** fue modificada en el código del Listado 12-10 para usar este valor y así crear nuevos archivos en la ruta correspondiente (dentro del directorio seleccionado por el usuario). Ahora, cada vez que un nuevo nombre de archivo es enviado desde el formulario, la ruta es agregada al nombre y el archivo es creado en el directorio actual.

Como ya explicamos, para mostrar la lista de entradas, debemos primero abrir el directorio a ser leído. Usando el método **`getDirectory()`** en la función **`mostrar()`**, el directorio actual (de acuerdo a la variable **`ruta`**) es abierto y una referencia a este directorio es enviada a la función **`leerdir()`** si la operación es exitosa. Esta función guarda la referencia en la variable **`dir`**, crea un nuevo objeto **`DirectoryReader`** para el directorio actual y obtiene la lista de entradas con el método **`readEntries()`**.

En **`leerdir(),`** funciones anónimas son usadas para mantener el código organizado y no superpoblar el entorno global. En primer lugar, **`createReader()`** crea un objeto **`DirectoryReader`** para el directorio representado por **`dir`**. Luego, una nueva función llamada **`leer()`** es creada dinámicamente para leer las entradas usando el método **`readEntries()`**. Este método lee las entradas por bloque, lo que significa que debemos llamarlo varias veces para asegurarnos de que todas las entradas disponibles en el directorio son leídas. La función **`leer()`** nos ayuda a lograr este propósito. El proceso es el siguiente: al final de la función **`leerdir()`**, la función **`leer()`** es llamada por primera vez. Dentro de la función **`leer()`** llamamos al método **`readEntries()`**. Este método usa otra función anónima en caso de éxito para recibir el objeto **`files`** y procesar su contenido (**`archivos`**). Si este objeto no está vacío, la función **`listar()`** es llamada para mostrar en pantalla las entradas leídas, y la función **`leer()`** es ejecutada nuevamente para obtener el siguiente bloque de entradas (la función se llama a sí misma una y otra vez hasta que ninguna entrada es retornada).

La función **`listar()`** está a cargo de imprimir la lista de entradas (archivos y directorios) en pantalla. Toma el objeto **`files`** y comprueba las características de cada entrada usando dos propiedades importantes de la interface **`Entry`**: **`isFile`** e **`isDirectory`**. Como sus nombres en inglés indican, estas propiedades contienen valores booleanos para informar si la entrada es un archivo o un directorio. Luego de que esta condición es controlada, la propiedad **`name`** es usada para mostrar información en la pantalla.

Existe una diferencia en cómo nuestra aplicación mostrará un archivo o un directorio en la pantalla. Cuando una entrada es detectada como directorio, es mostrada a través de un elemento **`<span>`** con un manejador de eventos **`onclick`** que llamará a la función **`cambiardir()`** si el usuario hace clic sobre el mismo. El propósito de esta función es declarar la nueva ruta actual para apuntar al directorio seleccionado. Recibe el nombre del directorio, agrega el directorio al valor de la variable **`ruta`** y llama a la función **`mostrar()`** para actualizar la información en pantalla (ahora deberían verse las entradas dentro del nuevo directorio seleccionado). Esta característica nos permite abrir directorios y ver su contenido con solo un clic del ratón, exactamente como una

explorador de archivos común y corriente haría.

Este ejemplo no contempla la posibilidad de retroceder en la estructura para ver el contenido de directorios padres. Para hacerlo, debemos aprovechar otro método provisto por la interface **Entry**:

getParent(función éxito, función error) Este método retorna un objeto **Entry** del directorio que contiene la entrada seleccionada. Una vez que obtenemos el objeto **Entry** podemos leer sus propiedades para obtener toda la información acerca del directorio padre de esa entrada en particular.

Cómo trabajar con el método **getParent()** es simple: supongamos que una estructura de directorios como **fotos/misvacaciones** fue creada y el usuario está listando el contenido de **misvacaciones** en este momento. Para regresar al directorio **fotos**, podríamos incluir un enlace en el documento HTML con un manejador de eventos **onclick** que llame a la función encargada de modificar la ruta actual para apuntar a esta nueva dirección (el directorio **fotos**). La función llamada al hacer clic sobre el enlace podría ser similar a la siguiente:

```
function volver(){
  dd.getDirectory(ruta,null,function(dir){
      dir.getParent(function(padre){
          ruta=padre.fullPath;
          mostrar();
        }, errores);
    },errores);
}
```

Listado 12-11. *Regresando al directorio padre.*

La función **volver()** en el Listado 12-11 cambia el valor de la variable **ruta** para apuntar al directorio padre del directorio actual. Lo primero que hacemos es obtener una referencia del directorio actual usando el método **getDirectory()**. Si la operación es exitosa, una función anónima es ejecutada. En esta función, el método **getParent()** es usado para encontrar el directorio padre del directorio referenciado por **dir** (el directorio actual). Si esta operación es exitosa, otra función anónima es ejecutada para recibir el objeto **padre** y declarar el valor de l a **ruta** actual igual al valor de la propiedad **fullPath** (esta propiedad contiene la ruta completa hacia el directorio padre). La función **mostrar()** es llamada al final del proceso para actualizar la información en pantalla (mostrar las entradas ubicadas en la nueva ruta).

Por supuesto, esta aplicación puede ser extremadamente enriquecida y mejorada, pero eso es algo que dejamos en sus manos.

Hágalo Usted Mismo: Agregue la función del Listado 12-11 al código del Listado 12-10 y cree un enlace en el documento HTML para llamar a esta función (por ejemplo, **volver**).

Manejando archivos

Ya mencionamos que la interface **Entry** incluye un grupo de propiedades y métodos para obtener información y operar archivos. Muchas de las propiedades disponibles ya fueron usadas en previos ejemplos. Ya aprovechamos las propiedades **isFile** e **isDirectory** para comprobar la clase de entrada, y también usamos los valores de **name**, **fullPath** y **filesystem** para mostrar información sobre la entrada en pantalla. El método **getParent()**, estudiado en el último código, es también parte de esta interface. Sin embargo, existen todavía algunos métodos más que son útiles para realizar operaciones comunes sobre archivos y directorios. Usando estos métodos podremos mover, copiar y eliminar entradas exactamente como en cualquier aplicación de escritorio:

moveTo(directorio, nombre, función éxito, función error) Este método mueve una entrada a una ubicación diferente en el Sistema de Archivos. Si el atributo **nombre** es provisto, el nombre de la entrada será cambiado a este valor.

copyTo(directorio, nombre, función éxito, función error) Este método genera una copia de una entrada en otra ubicación dentro del Sistema de Archivos. Si el atributo **nombre** es provisto, el nombre de la nueva entrada será cambiado a este valor.

remove() Este método elimina un archivo o un directorio vacío (para eliminar un directorio con contenido, debemos usar el método **removeRecursively()** presentado anteriormente).

Necesitaremos una nueva plantilla para probar estos métodos. Para simplificar los códigos, vamos a crear un

formulario con solo dos campos, uno para el origen y otro para el destino de cada operación:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>File API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Origen:<br><input type="text" name="origen" id="origen"
                                                    required></p>
      <p>Destino:<br><input type="text" name="destino"
                                       id="destino" required></p>
      <p><input type="button" name="boton" id="boton"
                                             value="Aceptar"></p>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 12-12: *nueva plantilla para operar con archivos*

Moviendo

El método **`moveTo()`** requiere un objeto **`Entry`** para el archivo y otro para el directorio en donde el archivo será movido. Por lo tanto, primero tenemos que crear una referencia al archivo que vamos a mover usando **`getFile()`**, luego obtenemos la referencia del directorio destino con **`getDirectory()`**, y finalmente aplicamos **`moveTo()`** con esta información:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', modificar, false);
  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                              creardd, errores);
}
function creardd(sistema){
  dd=sistema.root;
  ruta='';
  mostrar();
}
function errores(e){
  alert('Error: '+e.code);
}
function modificar(){
  var origen=document.getElementById('origen').value;
  var destino=document.getElementById('destino').value;

  dd.getFile(origen,null,function(archivo){
    dd.getDirectory(destino,null,function(dir){
        archivo.moveTo(dir,null,exito,errores);
      },errores);
    },errores);
}
function exito(){
  document.getElementById('origen').value='';
  document.getElementById('destino').value='';
```

```
  mostrar();
}
function mostrar(){
  cajadatos.innerHTML='';
  dd.getDirectory(ruta,null,leerdir,errores);
}
function leerdir(dir){
  var lector=dir.createReader();
  var leer=function(){
      lector.readEntries(function(archivos){
          if(archivos.length){
            listar(archivos);
            leer();
          }
        }, errores);
    }
  leer();
}
function listar(archivos){
  for(var i=0; i<archivos.length; i++) {
    if(archivos[i].isFile) {
      cajadatos.innerHTML+=archivos[i].name+'<br>';
    }else if(archivos[i].isDirectory){
      cajadatos.innerHTML+='<span onclick=
                          "cambiardir(\''+archivos[i].name+'\')"
            class="directorio">+'+archivos[i].name+'</span><br>';
    }
  }
}
function cambiardir(nuevaruta){
  ruta=ruta+nuevaruta+'/';
  mostrar();
}
window.addEventListener('load', iniciar, false);
```

Listado 12-13. *Moviendo archivos.*

En este último ejemplo, usamos funciones de códigos previos para crear o abrir nuestro Sistema de Archivos y mostrar el listado de entradas en pantalla. La única función nueva en el Listado 12-13 es **`modificar()`**. Esta función toma los valores de los campos del formulario **`origen`** y **`destino`** y los utiliza para abrir el archivo original y luego, si la operación es exitosa, abrir el directorio de destino. Si ambas operaciones son exitosas el método **`moveTo()`** es aplicado sobre el objeto **`file`** y el archivo es movido al directorio representado por **`dir`**. Si esta última operación es exitosa, la función **`exito()`** es llamada para vaciar los campos en el formulario y actualizar las entradas en pantalla ejecutando la función **`mostrar()`**.

> **Hágalo usted mismo:** Para probar este ejemplo necesita un archivo HTML con la plantilla del Listado 12-12, el archivo CSS usado desde el comienzo de este capítulo, y un archivo llamado **`file.js`** con el código del Listado 12-13 (recuerde subir los archivos a su servidor). Cree archivos y directorios usando códigos previos para tener entradas con las que trabajar. Utilice el formulario del último documento HTML para insertar los valores del archivo a ser movido (con la ruta completa desde la raíz) y el directorio en el cual el archivo será movido (si el directorio se encuentra en la raíz del Sistema de Archivos no necesita usar barras, solo su nombre).

Copiando

Por supuesto, la única diferencia entre el método **`moveTo()`** y el método **`copyTo()`** es que el último preserva el archivo original. Para usar el método **`copyTo()`**, solo debemos cambiar el nombre del método en el código del Listado 12-13. La función **`modificar()`** quedará como en el siguiente ejemplo:

```
function modificar(){
  var origen=document.getElementById('origen').value;
  var destino=document.getElementById('destino').value;
```

```
    dd.getFile(origen,null,function(archivo){
      dd.getDirectory(destino,null,function(dir){
          archivo.copyTo(dir,null,exito,errores);
        },errores);
      },errores);
  }
```

Listado 12-14. *Copiando archivos.*

Hágalo usted mismo: Reemplace la función **`modificar()`** del Listado 12-13 con esta última y abra la plantilla del Listado 12-12 para probar el código. Para copiar un archivo, debe repetir los pasos usados previamente para moverlo. Inserte la ruta del archivo a copiar en el campo **`origen`** y la ruta del directorio donde desea generar la copia en el campo **`destino`**.

Eliminando

Eliminar archivos y directorio es incluso más sencillo que mover y copiar. Todo lo que tenemos que hacer es obtener un objeto **`Entry`** del archivo o directorio que deseamos eliminar y aplicar el método **`remove()`** a esta referencia:

```
  function modificar(){
    var origen=document.getElementById('origen').value;
    var origen=ruta+origen;
    dd.getFile(origen,null,function(entrada){
        entrada.remove(exito,errores);
      },errores);
  }
```

Listado 12-15. *Eliminando archivos y directorios.*

El código del Listado 12-15 solo utiliza el valor del campo **`origen`** del formulario. Este valor, junto con el valor de la variable **`ruta`**, representará la ruta completa de la entrada que queremos eliminar. Usando este valor y el método **`getFile()`** creamos un objeto **`Entry`** para la entrada y luego aplicamos el método **`remove()`**.

Hágalo usted mismo: Reemplace la función **`modificar()`** en el código del Listado 12-13 con la nueva del Listado 12-15. Esta vez solo necesita ingresar el valor del campo **`origen`** para especificar el archivo a ser eliminado.

Para eliminar un directorio en lugar de un archivo, el objeto **`Entry`** debe ser creado para ese directorio usando **`getDirectory()`**, pero el método **`remove()`** trabaja exactamente igual sobre un tipo de entrada u otro. Sin embargo, debemos considerar una situación con respecto a la eliminación de directorios: si el directorio no está vacío, el método **`remove()`** retornará un error. Para eliminar un directorio y su contenido, todo al mismo tiempo, debemos usar otro método mencionado anteriormente llamado **`removeRecursively()`**:

```
  function modificar(){
    var destino=document.getElementById('destino').value;
    dd.getDirectory(destino,null,function(entrada){
        entrada.removeRecursively(exito,errores);
      },errores);
  }
```

Listado 12-16. *Eliminando directorios no vacíos.*

En la función del Listado 12-16 usamos el valor del campo **`destino`** para indicar el directorio a ser eliminado. El método **`removeRecursively()`** eliminará el directorio y su contenido en una sola ejecución y llamará a la función **`exito()`** si la operación es realizada con éxito.

Hágalo usted mismo: Las funciones **`modificar()`** presentadas en los Listados 12-14, 12-15 y 12-16 fueron construidas para reemplazar la misma función en el Listado 12-13. Para ejecutar estos ejemplos, utilice el código del Listado 12-13, reemplace la función **`modificar()`** por la que quiere probar y abra la plantilla del Listado 12-12 en su navegador. De acuerdo al método elegido, deberá ingresar uno o dos valores en el

formulario.

IMPORTANTE: Si tiene problemas para ejecutar estos ejemplos, le recomendamos usar la última versión disponible del navegador Chromium (*www.chromium.org*). Los códigos para esta parte de File API fueron también probados con éxito en Google Chrome.

12.4 Contenido de archivos

Además de la parte principal de API File y la extensión ya estudiada, existe otra especificación llamada API File: Writer. Esta extensión declara nuevas interfaces para escribir y agregar contenido a archivos. Trabaja junto con el resto de la API combinando métodos y compartiendo objetos para lograr su objetivo.

> **IMPORTANTE:** La integración entre todas las especificaciones involucradas en API despertó debate acerca de si algunas de las interfaces propuestas deberían ser movidas desde una API a otra. Para obtener información actualizada a este respecto, visite nuestro sitio web o el sitio de W3C en *www.w3.org*.

Escribiendo contenido

Para escribir contenido dentro de un archivo necesitamos crear un objeto **`FileWriter`**. Estos objetos son retornados por el método **`createWriter()`** de la interface **`FileEntry`**. Esta interface fue adicionada a la interface **`Entry`** y provee un total de dos métodos para trabajar con archivos:

createWriter(función éxito, función error) Este método retorna un objeto **`FileWriter`** asociado con la entrada seleccionada.

file(función éxito, función error) Este es un método que vamos a usar más adelante para leer el contenido del archivo. Crea un objeto **`File`** asociado con la entrada seleccionada (como el retornado por el elemento **`<input>`** o una operación arrastrar y soltar).

El objeto **`FileWriter`** retornado por el método **`createWriter()`** tiene sus propios métodos, propiedades y eventos para facilitar el proceso de agregar contenido a un archivo:

write(datos) Este es el método que escribe contenido dentro del archivo. El contenido a ser insertado es provisto por el atributo **`datos`** en forma de blob.

seek(desplazamiento) Este método establece la posición del archivo en la cual el contenido será escrito. El valor del atributo **`desplazamiento`** debe ser declarado en bytes.

truncate(tamaño) Este método cambia el tamaño del archivo de acuerdo al valor del atributo **`tamaño`** (en bytes).

position Esta propiedad retorna la posición actual en la cual la siguiente escritura ocurrirá. La posición será 0 para un nuevo archivo o diferente de 0 si algún contenido fue escrito dentro del archivo o el método **`seek()`** fue aplicado previamente.

length Esta propiedad retorna el largo del archivo.

writestart Este evento es disparado cuando el proceso de escritura comienza.

progress Este evento es disparado periódicamente para informar el progreso.

write Este evento es disparado cuando los datos han sido completamente escritos en el archivo.

abort Este evento es disparado si el proceso es abortado.

error Este evento es disparado si ocurre un error en el proceso.

writeend Este evento es disparado cuando el proceso termina.

Necesitamos crear un objeto más para preparar el contenido a ser agregado al archivo. El constructor **`BlobBuilder()`** retorna un objeto **`BlobBuilder`** con los siguientes métodos:

getBlob(tipo) Este método retorna el contenido del objeto **`BlobBuilder`** como un blob. Es útil para crear el blob que necesitamos usar con el método **`write()`**.

append(datos) Este método agrega el valor de **`datos`** al objeto **`BlobBuilder`**. El atributo **`datos`** puede ser un blob, un dato del tipo ArrayBuffer o simplemente texto.

El documento HTML del Listado 12-17 incorpora un segundo campo para insertar texto que representará el contenido del archivo. Será la plantilla utilizada en los próximos ejemplos:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>File API</title>
```

```
    <link rel="stylesheet" href="file.css">
    <script src="file.js"></script>
  </head>
  <body>
    <section id="cajaformulario">
      <form name="formulario">
        <p>Archivo:<br><input type="text" name="entrada"
                                   id="entrada" required></p>
        <p>Texto:<br><textarea name="texto" id="texto"
                                   required></textarea></p>
        <p><input type="button" name="boton" id="boton"
                                       value="Aceptar"></p>
      </form>
    </section>
    <section id="cajadatos">
      No hay información disponible
    </section>
  </body>
  </html>
```

Listado 12-17. *Formulario para ingresar el nombre del archivo y su contenido.*

Para la escritura del contenido abrimos el Sistema de Archivos, obtenemos o creamos el archivo con **getFile()** e insertamos contenido en su interior con los valores ingresados por el usuario. Con este fin, crearemos dos nuevas funciones: **escribirarchivo()** y **escribircontenido()**.

> **IMPORTANTE:** Hemos intentado mantener los códigos lo más simples posible por propósitos didácticos. Sin embargo, usted siempre puede aprovechar funciones anónimas para mantener todo dentro del mismo entorno (dentro de la misma función) o utilizar Programación Orientada a Objetos para implementaciones más poderosas y escalables.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', escribirarchivo, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                          creardd, errores);
}
function creardd(sistema){
  dd=sistema.root;
}
function errores(e){
  alert('Error: '+e.code);
}
function escribirarchivo(){
  var nombre=document.getElementById('entrada').value;
  dd.getFile(nombre, {create: true, exclusive:
                             false},function(entrada){
      entrada.createWriter(escribircontenido, errores);
    }, errores);
}
function escribircontenido(fileWriter) {
  var texto=document.getElementById('texto').value;
  fileWriter.onwriteend=exito;
  var blob=new WebKitBlobBuilder();
  blob.append(texto);
  fileWriter.write(blob.getBlob());
}
function exito(){
  document.getElementById('entrada').value='';
  document.getElementById('texto').value='';
  cajadatos.innerHTML='Hecho!';
}
window.addEventListener('load', iniciar, false);
```

Listado 12-18. *Escribiendo contenido.*

> **IMPORTANTE:** Del mismo modo que el método **`requestFileSystem()`**, Google Chrome ha agregado un prefijo al constructor **`BlobBuilder()`** en la implementación actual. Deberemos usar **`WebKitBlobBuilder()`** en éste y los siguientes ejemplos para probar nuestros códigos en este navegador. Como siempre, el método original podrá ser utilizado luego de que la etapa experimental sea finalizada.

Cuando el botón “Aceptar” es presionado, la información en los campos del formulario es procesada por las funciones **`escribirarchivo()`** y **`escribircontenido()`**. La función **`escribirarchivo()`** toma el valor del campo **`entrada`** y usa **`getFile()`** para abrir o crear el archivo si no existe. El objeto **`Entry`** retornado es usado por **`createWriter()`** para crear un objeto **`FileWriter`**. Si la operación es exitosa, este objeto es enviado a la función **`escribircontenido()`**.

La función **`escribircontenido()`** recibe el objeto **`FileWriter`** y, usando el valor del campo **`texto`**, escribe contenido dentro del archivo. El texto debe ser convertido en un blob antes de ser usado. Con este propósito, un objeto **`BlobBuilder`** es creado con el constructor **`BlobBuilder()`**, el texto es agregado a este objeto por el método **`append()`** y el contenido es recuperado como un blob usando **`getBlob()`**. Ahora la información se encuentra en el formato apropiado para ser escrita dentro del archivo usando **`write()`**.

Todo el proceso es asíncrono, por supuesto, lo que significa que el estado de la operación será contantemente informado a través de eventos. En la función **`escribircontenido()`**, solo escuchamos al evento **`writeend`** (usando el manejador de eventos **`onwriteend`**) para llamar a la función **`exito()`** y escribir el mensaje “Hecho!” en la pantalla cuando la operación es finalizada. Sin embargo, usted puede seguir el progreso o controlar los errores aprovechando el resto de los eventos disparados por el objeto **`FileWriter`**.

> **Hágalo usted mismo:** Copie la plantilla en el Listado 12-17 dentro de un nuevo archivo HTML (esta plantilla usa los mismos estilos CSS del Listado 12-2). Cree un archivo Javascript llamado **`file.js`** con el código del Listado 12-18. Abra el documento HTML en su navegador e inserte el nombre y el texto del archivo que quiere crear. Si el mensaje “Hecho!” aparece en pantalla, el proceso fue exitoso.

Agregando contenido

Debido a que no especificamos la posición en la cual el contenido debía ser escrito, el código previo simplemente escribirá el blob al comienzo del archivo. Para seleccionar una posición específica o agregar contenido al final de un archivo ya existente, es necesario usar previamente el método **`seek()`**.

```
function escribircontenido(fileWriter) {
  var texto=document.getElementById('texto').value;
  fileWriter.seek(fileWriter.length);
  fileWriter.onwriteend=exito;
  var blob=new WebKitBlobBuilder();
  blob.append(texto);
  fileWriter.write(blob.getBlob());
}
```

Listado 12-19. *Agregando contenido al final del archivo.*

La función del Listado 12-19 mejora la anterior función **`escribircontenido()`** incorporando un método **`seek()`** para mover la posición de escritura al final del archivo. De este modo, el contenido escrito por el método **`write()`** no sobrescribirá el contenido anterior.

Para calcular la posición del final del archivo en bytes, usamos la propiedad **`length`** mencionada anteriormente. El resto del código es exactamente el mismo que en el Listado 12-18.

> **Hágalo usted mismo:** Reemplace la función **`escribircontenido()`** del Listado 12-18 por la nueva en el Listado 12-19 y abra el archivo HTML en su navegador. Inserte en el formulario el mismo nombre del archivo creado usando el código previo y el texto que quiere agregar al mismo.

Leyendo contenido

Es momento de leer lo que acabamos de escribir. El proceso de lectura usa técnicas de la parte principal de API File, estudiada al comienzo de este capítulo. Vamos a usar el constructor **`FileReader()`** y métodos de lectura

como **readAsText()** para obtener el contenido del archivo.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', leerarchivo, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                              creardd, errores);
}
function creardd(sistema){
  dd=sistema.root;
}
function errores(e){
  alert('Error: '+e.code);
}
function leerarchivo(){
  var nombre=document.getElementById('entrada').value;
  dd.getFile(nombre, {create: false}, function(entrada) {
    entrada.file(leercontenido, errores);
  }, errores);
}
function leercontenido(archivo){
  cajadatos.innerHTML='Nombre: '+archivo.name+'<br>';
  cajadatos.innerHTML+='Tipo: '+archivo.type+'<br>';
  cajadatos.innerHTML+='Tamaño: '+archivo.size+' bytes<br>';

  var lector=new FileReader();
  lector.onload=exito;
  lector.readAsText(archivo);
}
function exito(e){
  var resultado=e.target.result;
  document.getElementById('entrada').value='';
  cajadatos.innerHTML+='Contenido: '+resultado;
}
window.addEventListener('load', iniciar, false);
```

Listado 12-20. *Leyendo el contenido de un archivo en el Sistema de Archivos.*

Los métodos provistos por la interface **FileReader** para leer el contenido de un archivo, como **readAsText()**, requieren un blob o un objeto **File** como atributo. El objeto **File** representa el archivo a ser leído y es generado por el elemento **<input>** o una operación arrastrar y soltar. Como dijimos anteriormente, la interface **FileEntry** ofrece la opción de crear esta clase de objetos utilizando un método llamado **file()**.

Cuando el botón “Aceptar” es presionado, la función **leerarchivo()** toma el valor del campo **entrada** del formulario y abre el archivo con ese nombre usando **getFile()**. El objeto **Entry** retornado por este método es representado por la variable **entrada** y usado para generar el objeto **File** con el método **file()**.

Debido a que el objeto **File** obtenido de este modo es exactamente el mismo generado por el elemento **<input>** o la operación arrastrar y soltar, todas las mismas propiedades usadas antes están disponibles y podemos mostrar información básica acerca del archivo incluso antes de que el proceso de lectura del contenido comience. En la función **leercontenido()**, los valores de estas propiedades son mostrados en pantalla y el contenido del archivo es leído.

El proceso de lectura es una copia exacta del código del Listado 12-3: el objeto **FileReader** es creado con el constructor **FileReader()**, el manejador de eventos **onload** es registrado para llamar a la función **exito()** cuando el proceso es finalizado, y el contenido del archivo es finalmente leído por el método **readAsText()**.

En la función **exito()**, en lugar de imprimir un mensaje como hicimos previamente, el contenido del archivo es mostrado en pantalla. Para hacer esto, tomamos el valor de la propiedad **result** perteneciente al objeto **FileReader** y lo declaramos como contenido del elemento **cajadatos**.

Hágalo Usted Mismo: El código en el Listado 12-20 utiliza solo el valor del campo **entrada** (no necesita escribir un contenido para el archivo, solo ingresar su nombre). Abra el archivo HTML con la última plantilla en su navegador e inserte el nombre del archivo que quiere leer. Debe ser un archivo que usted ya creó

usando códigos previos o el sistema retornará un mensaje de error (**`create: false`**). Si el nombre de archivo es correcto, la información sobre este archivo y su contenido serán mostrados en pantalla.

12.5 Sistema de archivos de la vida real

Siempre es bueno estudiar un caso de la vida real que nos permita entender el potencial de los conceptos aprendidos. Para finalizar este capítulo, vamos a crear una aplicación que combina varias técnicas de API File con las posibilidades de manipulación de imágenes ofrecida por API Canvas.

Este ejemplo toma múltiples archivos de imagen y dibuja estas imágenes en el lienzo en una posición seleccionada al azar. Cada cambio efectuado en el lienzo es grabado en un archivo para lecturas posteriores, por lo tanto cada vez que acceda a la aplicación el último trabajo realizado sobre el lienzo será mostrado en pantalla.

El documento HTML que vamos a crear es similar a la primera plantilla utilizada en este capítulo. Sin embargo, esta vez incluimos un elemento **<canvas>** dentro del elemento **cajadatos**:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>File API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Imágenes:<br><input type="file" name="archivos"
                                  id="archivos" multiple></p>
    </form>
  </section>
  <section id="cajadatos">
    <canvas id="lienzo" width="500" height="350"></canvas>
  </section>
</body>
</html>
```

Listado 12-21. *Nueva plantilla con el elemento* <canvas>.

El código de este ejemplo incluye métodos y técnicas de programación con las que ya está familiarizado, pero la combinación de especificaciones puede resultar confusa al principio. Veamos primero el código y analicemos luego cada parte paso a paso:

```
function iniciar(){
  var elemento=document.getElementById('lienzo');
  lienzo=elemento.getContext('2d');
  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', procesar, false);
  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
                                              creardd, errores);
}
function creardd(sistema){
  dd=sistema.root;
  cargarlienzo();
}
function errores(e){
  alert('Error: '+e.code);
}
function procesar(e){
  var archivos=e.target.files;
  for(var f=0;f<archivos.length;f++){
    var archivo=archivos[f];
    if(archivo.type.match(/image.*/i)){
      var lector=new FileReader();
      lector.onload=mostrar;
      lector.readAsDataURL(archivo);
    }
```

```
    }
  }
  function mostrar(e){
    var resultado=e.target.result;
    var imagen=new Image();
    imagen.src=resultado;
    imagen.addEventListener("load", function(){
      var x=Math.floor(Math.random()*451);
      var y=Math.floor(Math.random()*301);
      lienzo.drawImage(imagen,x,y,100,100);
      grabarlienzo();
    }, false);
  }
  function cargarlienzo(){
    dd.getFile('lienzo.dat', {create: false}, function(entrada) {
      entrada.file(function(archivo){
          var lector=new FileReader();
          lector.onload=function(e){
              var imagen=new Image();
              imagen.src=e.target.result;
              imagen.addEventListener("load", function(){
                  lienzo.drawImage(imagen,0,0);
                }, false);
            };
          lector.readAsBinaryString(archivo);
        }, errores);
    }, errores);
  }
  function grabarlienzo(){
    var elemento=document.getElementById('lienzo');
    var info=elemento.toDataURL();
    dd.getFile('lienzo.dat', {create: true, exclusive: false},
                                            function(entrada) {
        entrada.createWriter(function(fileWriter){
            var blob=new WebKitBlobBuilder();
            blob.append(info);
            fileWriter.write(blob.getBlob());
          }, errores);
      }, errores);
  }
  window.addEventListener('load', iniciar, false);
```

Listado 12-22. *Combinando API File y API Canvas.*

En este ejemplo trabajamos con dos APIs: API File (con sus extensiones) y API Canvas. En la función **iniciar()**, ambas APIs son inicializadas. El contexto para el lienzo es generado primero usando **getContext()**, y el Sistema de Archivos es solicitado después por el método **requestFileSystem()**.

Como siempre, una vez que el Sistema de Archivos está listo, la función **creardd()** es llamada y la variable **dd** es inicializada en esta función con una referencia al directorio raíz del Sistema de Archivos. Esta vez una llamada a una nueva función fue agregada al final de **creardd()** con el propósito de cargar el archivo conteniendo la imagen generada por la aplicación la última vez que fue ejecutada.

Veamos en primer lugar cómo la imagen grabada en el archivo mencionado es construida. Cuando el usuario selecciona un nuevo archivo de imagen desde el formulario, el evento **change** es disparado por el elemento **<input>** y la función **procesar()** es llamada. Esta función toma los archivos enviados por el formulario, extrae cada archivo del objeto **File** recibido, controla si se trata de una imagen o no, y en caso positivo lee el contenido de cada entrada con el método **readAsDataURL()**, retornando un valor en formato data:url.

Como puede ver, cada archivo es leído por la función **procesar()**, uno a la vez. Cada vez que una de estas operaciones es exitosa, el evento **load** es disparado y la función **mostrar()** es ejecutada.

La función **mostrar()** toma los datos del objeto **lector** (recibidos a través del evento), crea un objeto **imagen** con el constructor **Image()**, y asigna los datos leídos previamente como la fuente de esa imagen con la línea **imagen.src=resultado**.

Cuando trabajamos con imágenes siempre debemos considerar el tiempo que la imagen tarda en ser cargada en memoria. Por esta razón, luego de declarar la nueva fuente del objeto **imagen** agregamos una

escucha para el evento **load** que nos permitirá procesar la imagen solo cuando fue completamente cargada. Cuando este evento es disparado, la función anónima declarada para responder al evento en el método **addEventListener()** es ejecutada. Esta función calcula una posición al azar para la imagen dentro del lienzo y la dibuja usando el método **drawImage()**. La imagen es reducida por este método a un tamaño fijo de 100x100 pixeles, sin importar el tamaño original (estudie la función **mostrar()** en el Listado 12-22 para entender cómo funciona todo el proceso).

Luego de que las imágenes seleccionadas son dibujadas, la función **grabarlienzo()** es llamada. Esta función se encargará de grabar el estado del lienzo cada vez que es modificado, permitiendo a la aplicación recuperar el último trabajo realizado la próxima vez que es ejecutada. El método de API Canvas llamado **toDataURL()** es usado para retornar el contenido del lienzo como data:url. Para procesar estos datos, varias operaciones son realizadas dentro de **grabarlienzo()**. Primero, los datos en formato data:url son almacenados dentro de la variable **info**. Luego, el archivo **lienzo.dat** es creado (si aún no existe) o abierto con **getFile()**. Si esta operación es exitosa, la entrada es tomada por una función anónima y el objeto **FileWriter** es creado por el método **createWriter()**. Si esta operación es exitosa, este método también llama a una función anónima donde el valor de la variable **info** (los datos sobre el estado actual del lienzo) son agregados a un objeto **BlobBuilder** y el blob dentro del mismo es finalmente escrito dentro del archivo **lienzo.dat** por medio de **write()**.

> **IMPORTANTE:** En esta oportunidad no escuchamos ningún evento del objeto **FileWriter** porque no hay nada que necesitemos hacer en caso de éxito o error. Sin embargo, usted siempre puede aprovechar los eventos para reportar el estado de la operación en la pantalla o tener control total sobre cada parte del proceso.

Bien, es momento de volver a la función **cargarlienzo()**. Como ya mencionamos, esta función es llamada por la función **creardd()** tan pronto como la aplicación es cargada. Tiene el propósito de leer el archivo con el trabajo anterior y dibujarlo en pantalla. A este punto usted ya sabe de qué archivo estamos hablando y cómo es generado, veamos entonces cómo esta función restaura el último trabajo realizado sobre el lienzo.

La función **cargarlienzo()** carga el archivo **lienzo.dat** para obtener los datos en formato data:url generados la última vez que el lienzo fue modificado. Si el archivo no existe, el método **getFile()** retornará un error, pero cuando es encontrado el método ejecuta una función anónima que tomará la entrada y usará el método **file()** para generar un objeto **File** con estos datos. Este método, si es exitoso, también ejecuta una función anónima para leer el archivo y obtener su contenido como datos binarios usando **readAsBinaryString()**. El contenido obtenido, como ya sabemos, es una cadena de texto en formato data:url que debe ser asignado como fuente de una imagen antes de ser dibujado en el lienzo. Por este motivo, lo que hacemos dentro de la función anónima llamada por el evento **load** una vez que el archivo es completamente cargado, es crear un objeto **imagen**, declarar los datos obtenidos como la fuente de esta imagen, y (cuando la imagen es completamente cargada) dibujarla en el lienzo con **drawImage()**.

El resultado obtenido por esta pequeña pero interesante aplicación es sencillo: las imágenes seleccionadas desde el elemento **<input>** son dibujadas en el lienzo en una posición al azar y el estado del lienzo es preservado en un archivo. Si el navegador es cerrado, no importa por cuánto tiempo, la próxima vez que la aplicación es usada el archivo es leído, el estado previo del lienzo es restaurado y nuestro último trabajo sigue ahí, como si nunca lo hubiésemos abandonado. No es realmente un ejemplo útil, pero se puede apreciar su potencial.

> **Hágalo Usted Mismo:** Usando la API Drag and Drop puede arrastrar y soltar archivos de imagen dentro del lienzo en lugar de cargar las imágenes desde el elemento **<input>**. Intente combinar el código del Listado 12-22 con algunos códigos del Capítulo 8 para integrar estas APIs.

12.6 Referencia rápida

Del mismo modo que la API IndexedDB, las características de API File y sus extensiones fueron organizadas en interfaces. Cada interface provee métodos, propiedades y eventos que trabajan combinados con el resto para ofrecer diferentes alternativas con las que crear, leer y procesar archivos. En esta referencia rápida vamos a presentar todas las características estudiadas en este capítulo en un orden acorde a esta organización oficial.

> **IMPORTANTE:** Las descripciones presentadas en esta referencia rápida solo muestran los aspectos más relevantes de cada interface. Para estudiar la especificación completa, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Interface Blob (API File)

Esta interface provee propiedades y métodos para operar con blobs. Es heredada por la interface File.

size Esta propiedad retorna el tamaño del blob o el archivo en bytes.

type Esta propiedad retorna el tipo de medio dentro de un blob o archivo.

slice(comienzo, largo, tipo) Este método retorna la parte del blob o archivo indicada por los valores en bytes de los atributos `comienzo` y `largo`.

Interface File (API File)

Esta interface es una extensión de la interface Blob para procesar archivos.

name Esta propiedad retorna el nombre del archivo.

Interface FileReader (API File)

Esta interface provee métodos, propiedades y eventos para cargar blobs y archivos en memoria.

readAsArrayBuffer(archivo) Este método retorna el contenido de blobs o archivos en el formato ArrayBuffer.

readAsBinaryString(archivo) Este método retorna el contenido de blobs o archivos como una cadena binaria.

readAsText(archivo) Este método interpreta el contenido de blobs o archivos y lo retorna en formato texto.

readAsDataURL(archivo) Este método retorna el contenido de blobs o archivos en el formato data:url.

abort() Este método aborta el proceso de lectura.

result Esta propiedad representa los datos retornados por los métodos de lectura.

loadstart Este evento es disparado cuando la lectura comienza.

progress Este evento es disparado periódicamente para reportar el estado del proceso de lectura.

load Este evento es disparado cuando el proceso de lectura es finalizado.

abort Este evento es disparado cuando el proceso de lectura es abortado.

error Este evento es disparado cuando un error ocurre en el proceso.

loadend Este evento es disparado cuando la carga del archivo es finalizada, haya sido el proceso exitoso o no.

Interface LocalFileSystem (API File: Directories and System)

Esta interface es provista para iniciar un Sistema de Archivos para la aplicación.

requestFileSystem(tipo, tamaño, función éxito, función error) Este método solicita la inicialización de un Sistema de Archivos configurado de acuerdo a los valores de sus atributos. El atributo `tipo` puede recibir dos valores diferentes: `TEMPORARY` (temporario) o `PERSISTENT` (persistente). El tamaño debe ser

especificado en bytes.

Interface FileSystem (API File: Directories and System)

Esta interface provee información acerca del Sistema de Archivos.

name Esta propiedad retorna el nombre del Sistema de Archivos.

root Esta propiedad retorna una referencia el directorio raíz del Sistema de Archivos.

Interface Entry (API File: Directories and System)

Esta interface provee métodos y propiedades para procesar entradas (archivos y directorios) en el Sistema de Archivos.

isFile Esta propiedad es un valor booleano que indica si la entrada es un archivo o no.

isDirectory Esta propiedad es un valor booleano que indica si la entrada es un directorio o no.

name Esta propiedad retorna el nombre de la entrada.

fullPath Esta propiedad retorna la ruta completa de la entrada desde el directorio raíz del Sistema de Archivos.

filesystem Esta propiedad contiene una referencia al Sistema de Archivos.

moveTo(directorio, nombre, función éxito, función error) Este método mueve una entrada a una ubicación diferente dentro del Sistema de Archivos. El atributo `directorio` representa el directorio dentro del cual la entrada será movida. El atributo `nombre`, si es especificado, cambia el nombre de la entrada en la nueva ubicación.

copyTo(directorio, nombre, función éxito, función error) Este método genera una copia de la entrada dentro del Sistema de Archivos. El atributo `directorio` representa el directorio dentro del cual la copia de la entrada será creada. El atributo `nombre`, si es especificado, cambia el nombre de la copia.

remove(función éxito, función error) Este método elimina un archivo o un directorio vacío.

getParent(función éxito, función error) Este método retorna el objeto `DirectoryEntry` padre de la entrada seleccionada.

Interface DirectoryEntry (API File: Directories and System)

Esta interface provee métodos para crear y leer archivos y directorios.

createReader() Este método crear un objeto `DirectoryReader` para leer entradas.

getFile(ruta, opciones, función éxito, función error) Este método crea o lee el archivo indicado por el atributo `ruta`. El atributo `opciones` es declarado por dos banderas: `create` (crear) y `exclusive` (exclusivo). La primera indica si el archivo será creado o no, y la segunda, cuando es declarada como `true` (verdadero), fuerza al método a retornar un error si el archivo ya existe.

getDirectory(ruta, opciones, función éxito, función error) Este método crea o lee el directorio indicado por el atributo `ruta`. El atributo `opciones` es declarado por dos banderas: `create` (crear) y `exclusive` (exclusivo). La primera indica si el directorio será creado o no, y la segunda, cuando es declarada como `true` (verdadero), fuerza al método a retornar un error si el directorio ya existe.

removeRecursively(función éxito, función error) Este método elimina un directorio y todo su contenido.

Interface DirectoryReader (API File: Directories and System)

Esta interface ofrece la posibilidad de obtener una lista de entradas en un directorio específico.

readEntries(función éxito, función error) Este método lee un bloque de entradas desde el directorio seleccionado. Retorna el valor `null` si no se encuentran más entradas.

Interface FileEntry (API File: Directories and System)

Esta interface provee métodos para obtener un objeto **`File`** para un archivo específico y un objeto **`FileWriter`** para poder agregar contenido al mismo.

createWriter(función éxito, función error) Este método crea un objeto FileWriter para escribir contenido dentro de un archivo.

file(función éxito, función error) Este método retorna un objeto File que representa el archivo seleccionado.

Interface BlobBuilder (API File: Writer)

Esta interface provee métodos para trabajar con objetos blob.

getBlob(tipo) Este método retorna el contenido de un objeto blob como un blob.

append(datos) Este método agrega datos a un objeto blob. La interface provee tres métodos **`append()`** diferentes para agregar datos en forma de texto, blob, o como un ArrayBuffer.

Interface FileWriter (API File: Writer)

La interface FileWriter es una expansión de la interface FileSaver. La última no es descripta aquí, pero los eventos listados debajo son parte de ella.

position Este propiedad retorna la posición actual en la cual se realizará la siguiente escritura.

length Esta propiedad retorna el largo del archivo en bytes.

write(blob) Este método escribe contenido en un archivo.

seek(desplazamiento) Este método especifica una nueva posición en la cual se realizará la siguiente escritura.

truncate(tamaño) Este método cambia el largo del archivo al valor del atributo **`tamaño`** (en bytes).

writestart Este evento es disparado cuando la escritura comienza.

progress Este evento es disparado periódicamente para informar sobre el estado del proceso de escritura.

write Este evento es disparado cuando el proceso de escritura es finalizado.

abort ste evento es disparado cuando el proceso de escritura es abortado.

error Este evento es disparado si ocurre un error en el proceso de escritura.

writeend Este evento es disparado cuando la solicitud es finalizada, haya sido exitosa o no.

Interface FileError (API File y extensiones)

Varios métodos en esta API retornan un valor a través de una función para indicar errores en el proceso. Este valor puede ser comparado con la siguiente lista para encontrar el error correspondiente:

NOT_FOUND_ERR - valor 1.

SECURITY_ERR - valor 2.

ABORT_ERR - valor 3.

NOT_READABLE_ERR - valor 4.

ENCODING_ERR - valor 5

NO_MODIFICATION_ALLOWED_ERR - valor 6.

INVALID_STATE_ERR - valor 7.

SYNTAX_ERR - valor 8.

INVALID_MODIFICATION_ERR - valor 9.

QUOTA_EXCEEDED_ERR - valor 10.

TYPE_MISMATCH_ERR - valor 11.

PATH_EXISTS_ERR - valor 12.

Capítulo 13
API Communication

13.1 Ajax nivel 2

Esta es la primera parte de lo que llamamos API Communication. Lo que extra oficialmente es conocido como API Communication es en realidad un grupo de APIs compuesto por XMLHttpRequest Level 2, Cross Document Messaging (API Web Messaging), y Web Sockets (API WebSocket). La primera de estas tres tecnologías de comunicación es una mejora del viejo objeto XMLHttpRequest usado extensamente hasta estos días para comunicarse con servidores y construir aplicaciones Ajax.

El nivel 2 de XMLHttpRequest incorpora nuevas características como comunicación con múltiples orígenes y eventos para controlar la evolución de las solicitudes. Estas mejoras simplifican códigos y ofrecen nuevas opciones, como interactuar con diferentes servidores desde la misma aplicación o trabajar con pequeñas trozos de datos en lugar de archivos enteros, por nombrar unas pocas.

El elemento más importante de esta API es, por supuesto, el objeto XMLHttpRequest. Un constructor fue especificado para crearlo:

XMLHttpRequest() Este constructor retorna un objeto XMLHttpRequest por medio del cual podemos comenzar una solicitud y escuchar eventos para controlar el proceso de comunicación.

El objeto creado por **`XMLHttpRequest()`** cuenta con importantes métodos para iniciar y controlar la solicitud:

open(método, url, asíncrono) Este método configura una solicitud pendiente. El atributo **`método`** declara el tipo de método HTTP usado para abrir la conexión (**`GET`** o **`POST`**). El atributo **`url`** declara la ubicación del código que va a procesar la solicitud. Y **`asíncrono`** es un valor booleano para declarar si la conexión será síncrona (**`false`**) o asíncrona (**`true`**). De ser necesario, el método también puede incluir valores especificando el nombre de usuario y la clave.

send(datos) Este es el método que inicia la solicitud. Existen varias versiones de este método en un objeto XMLHttpRequest para procesar diferentes tipos de datos. El atributo **`datos`** puede ser omitido, declarado como un ArrayBuffer, un blob, un documento, una cadena de texto, o como FormData (ya estudiaremos este nuevo tipo de datos más adelante).

abort() Este método cancela la solicitud.

Obteniendo datos

Comencemos construyendo un ejemplo que obtiene el contenido de un archivo de texto en el servidor usando el método **`GET`**. Vamos a necesitar un nuevo documento HTML con un botón para iniciar la solicitud:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p><input type="button" name="boton" id="boton"
                                         value="Aceptar"></p>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 13-1. *Plantilla para solicitudes Ajax.*

Para hacer los códigos tan simples como sea posible mantuvimos la misma estructura HTML usada previamente y aplicamos solo los siguientes estilos por propósitos visuales:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 13-2. *Estilos para dar forma a las cajas en pantalla .*

Hágalo usted mismo: Necesita crear un archivo HTML con la plantilla del Listado 13-1 y un archivo CSS llamado **ajax.css** con las reglas del Listado 13-2. Para poder probar estos ejemplos, deberá subir los archivos a su servidor, incluyendo el archivo Javascript y el que recibe la solicitud. Vamos a proveer más instrucciones en cada ejemplo.

El código para este primer ejemplo leerá un archivo en el servidor y mostrará su contenido en pantalla. Ningún dato es enviado al servidor; solo tenemos que hacer una solicitud **GET** y mostrar la información obtenida en respuesta:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  var boton=document.getElementById('boton');
  boton.addEventListener('click', leer, false);
}
function leer(){
  var url="texto.txt";
  var solicitud=new XMLHttpRequest();
  solicitud.addEventListener('load',mostrar,false);
  solicitud.open("GET", url, true);
  solicitud.send(null);
}
function mostrar(e){
  cajadatos.innerHTML=e.target.responseText;
}
window.addEventListener('load', iniciar, false);
```

Listado 13-3. *Leyendo un archivo en el servidor.*

En el código del Listado 13-3 incluimos nuestra típica función **iniciar()**. Esta función es llamada cuando el documento es cargado. Lo que hace en este caso es simplemente crear una referencia al elemento **cajadatos** y agrega una escucha para el evento **click** en el botón del formulario.

Cuando el botón “Aceptar” es presionado, la función **leer()** es ejecutada. Aquí podemos ver en acción todos los métodos estudiados previamente. Primero, la URL del archivo que será leído es declarada. No explicamos todavía cómo hacer solicitudes a diferentes servidores, por lo que este archivo deberá estar ubicado en el mismo dominio que el código Javascript (y en este ejemplo, también en el mismo directorio). En el siguiente paso, el objeto es creado por el constructor **XMLHttpRequest()** y asignado a la variable **solicitud**. Esta variable es usada luego para agregar una escucha para el evento **load** e iniciar la solicitud usando los métodos **open()** y **send()**. Debido a que ningún dato será enviado en esta solicitud, un valor **null** fue declarado en el método **send()**. En el método **open()**, en cambio, declaramos la solicitud como del tipo **GET**, la URL del archivo a ser leído, y el tipo de operación (**true** para asíncrona).

Una operación asíncrona significa que el navegador continuará procesando el resto del código mientras espera que el archivo termine de ser descargado desde el servidor. El final de la operación será informado a

través del método **load**. Cuando este evento es disparado, la función **mostrar()** es llamada. Esta función reemplaza el contenido del elemento **cajadatos** por el valor de la propiedad **responseText**, y el proceso finaliza.

Hágalo usted mismo: Para probar este ejemplo, cree un archivo de texto llamado **texto.txt** y agregue algún texto al mismo. Suba este archivo y el resto de los archivos creados con los códigos 13-1, 13-2 y 13-3 a su servidor y abra el documento HTML en su navegador. Luego de hacer clic sobre el botón "Aceptar", el contenido del archivo de texto es mostrado en pantalla.

IMPORTANTE: Cuando la respuesta es procesada usando **innerHTML**, los códigos HTML y Javascript son procesados. Por razones de seguridad siempre es mejor usar **innerText** en su lugar. Usted deberá tomar la decisión de utilizar uno u otro de acuerdo a las características de su aplicación.

Propiedades response

Existen tres tipos diferentes de propiedades **response** que podemos usar para obtener la información retornada por la solicitud:

response Esta es una propiedad de propósito general. Retorna la respuesta de la solicitud de acuerdo al valor del atributo **responseType**.

responseText Esta propiedad retorna la respuesta a la solicitud en formato texto.

responseXML Esta propiedad retorna la respuesta a la solicitud como si fuera un documento XML.

Eventos

Además de **load**, la especificación incluye otros eventos para el objeto XMLHttpRequest:

loadstart Este evento es disparado cuando la solicitud comienza.

progress Este evento es disparado periódicamente mientras se envían o descargan datos.

abort Este evento es disparado cuando la solicitud es abortada.

error Este evento es disparado cuando un error ocurre durante el procesamiento de la solicitud.

load Este evento es disparado cuando la solicitud ha sido completada.

timeout Si un valor para **timeout** ha sido especificado, este evento será disparado cuando la solicitud no pueda ser completada en el período de tiempo determinado.

loadend Este evento es disparado cuando la solicitud ha sido completada (sin considerar si el proceso fue exitoso o no).

Quizás el evento más atractivo de todos sea **progress**. Este evento es disparado aproximadamente cada 50 milisegundos para informar acerca del estado de la solicitud. Gracias a este evento podemos informar al usuario sobre cada paso del proceso y crear aplicaciones de comunicación profesionales.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  var boton=document.getElementById('boton');
  boton.addEventListener('click', leer, false);
}
function leer(){
  var url="trailer.ogg";
  var solicitud=new XMLHttpRequest();
  solicitud.addEventListener('loadstart',comenzar,false);
  solicitud.addEventListener('progress',estado,false);
  solicitud.addEventListener('load',mostrar,false);
  solicitud.open("GET", url, true);
  solicitud.send(null);
}
function comenzar(){
  cajadatos.innerHTML='<progress value="0" max="100">0%</progress>';
}
```

```
function estado(e){
  if(e.lengthComputable){
    var por=parseInt(e.loaded/e.total*100);
    var barraprogreso=cajadatos.querySelector("progress");
    barraprogreso.value=por;
    barraprogreso.innerHTML=por+'%';
  }
}
function mostrar(e){
  cajadatos.innerHTML='Terminado';
}
window.addEventListener('load', iniciar, false);
```

Listado 13-4. *Informando el progreso de la solicitud.*

En el Listado 13-4, el código usa tres eventos, **loadstart**, **progress** y **load**, para controlar la solicitud. El evento **loadstart** llama a la función **comenzar()** y la barra de progreso es mostrada en la pantalla por primera vez. Mientras el archivo es descargado, el evento **progress** ejecutará la función **estado()** constantemente. Esta función informa sobre el progreso de la operación a través del elemento **<progress>** creado anteriormente y el valor de las propiedades ofrecidas por el evento.

Finalmente, cuando el archivo es completamente descargado, el evento **load** es disparado y la función **mostrar()** imprime el texto "Terminado" en la pantalla.

IMPORTANTE: En nuestro ejemplo utilizamos **innerHTML** para agregar un nuevo elemento **<progress>** al documento. Esta no es una práctica recomendada pero es útil y conveniente por razones didácticas. Normalmente los elementos son agregados al documento usando el método Javascript **createElement()** junto con **appendChild()**.

El evento **progress** es declarado por la especificación en la interface **ProgressEvent**. Esta interface es común a cada API e incluye tres valiosas propiedades para retornar información sobre el proceso que es monitoreado por el evento:

lengthComputable Esta propiedad retorna **true** (verdadero) cuando el progreso puede ser calculado o **false** (falso) en caso contrario. Lo usamos en nuestro ejemplo para estar seguros de que los valores de las propiedades restantes son reales y válidos.

loaded Esta propiedad retorna el total de bytes ya subidos o descargados.

total Este propiedad retorna el tamaño total de los datos que están siendo subidos o descargados.

IMPORTANTE: Dependiendo de su conexión a Internet, para ver cómo la barra de progreso trabaja, es posible que deba usar archivos extensos. En el código del Listado 13-4, declaramos la URL como el nombre del video usado en el Capítulo 5 para trabajar con la API de medios. Puede usar sus propios archivos o descargar este video en: *WWW.minkbooks.com/content/trailer.ogg.*

Enviando datos

Hasta el momento hemos leído información desde el servidor, pero no hemos enviado ningún dato o incluso usado otro método HTTP además de **GET**. En el siguiente ejemplo vamos a trabajar con el método **POST** y un nuevo objeto que nos permite enviar información usando formularios virtuales.

En el ejemplo anterior no mencionamos cómo enviar datos con el método **GET** porque, como siempre, es tan simple como agregar los valores a la URL. Solo tenemos que crear una ruta para la variable **url** como **textfile.txt?val1=1&val2=2** y los valores especificados serán enviados junto con la consulta. Los atributos **val1** y **val2** de este ejemplo serán leídos como variables **GET** del lado del servidor. Por supuesto, un archivo de texto no puede procesar esta información, por lo que normalmente deberemos recibir los datos usando un archivo programado en PHP, o en cualquier otro lenguaje de procesamiento en el servidor. Las solicitudes **POST**, por otro lado, no son tan simples.

Como ya seguramente conoce, una solicitud **POST** incluye toda la información enviada por un método **GET** pero también el cuerpo del mensaje. El cuerpo del mensaje representa cualquier información de cualquier tipo y tamaño a ser enviada. Un formulario HTML es normalmente la mejor manera de proveer esta información, pero para aplicaciones dinámicas esta no es probablemente la mejor opción o la más apropiada. Para resolver este problema, la API incluye la interface **FormData**. Esta interface sencilla tiene solo un constructor y un método con el que obtener y trabajar sobre objetos **FormData**.

FormData() Este constructor retorna una objeto **`FormData`** usado luego por el método **`send()`** para enviar información.

append(nombre, valor) Este método agrega datos al objeto **`FormData`**. Toma un par clave/valor como atributos. El atributo **`valor`** puede ser una cadena de texto o un blob. Los datos retornados representan un campo de formulario.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);
}
function enviar(){
  var datos=new FormData();
  datos.append('nombre','Juan');
  datos.append('apellido','Perez');

  var url="procesar.php";
  var solicitud=new XMLHttpRequest();
  solicitud.addEventListener('load', mostrar, false);
  solicitud.open("POST", url, true);
  solicitud.send(datos);
}
function mostrar(e){
  cajadatos.innerHTML=e.target.responseText;
}
window.addEventListener('load', iniciar, false);
```

Listado 13-5. *Enviando un formulario virtual.*

Cuando la información es enviada al servidor, es con el propósito de procesarla y producir un resultado. Normalmente este resultado es almacenado en el servidor y algunos datos son retornados como respuesta al usuario. En el ejemplo del Listado 13-5, los datos son enviados al archivo **`procesar.php`** y la respuesta de este código es mostrada en pantalla.

Para probar este ejemplo, el archivo **`procesar.php`** deberá imprimir los valores recibidos con un código similar al siguiente:

```
<?PHP
  print('Su nombre es: '.$_POST['nombre'].'<br>');
  print('Su apellido es: '.$_POST['apellido']);
?>
```

Listado 13-6. *Respuesta simple a una solicitud `POST` (`procesar.php`).*

Veamos en primer lugar cómo la información fue preparada para ser enviada. En la función **`send()`** del Listado 13-5, el constructor **`FormData()`** es invocado y el objeto **`FormData`** retornado es almacenado en la variable **`datos`**. Dos pares clave/valor son agregados luego a este objeto con los nombres **`nombre`** y **`apellido`** usando el método **`append()`**. Estos valores representarán campos de formulario.

La inicialización de la solicitud es exactamente la misma que en códigos previos, excepto que esta vez el primer atributo del método **`open()`** es **`POST`** en lugar de **`GET`**, y el atributo del método **`send()`** es el objeto **`datos`** que acabamos de construir y no un valor nulo (**`null`**), como usamos anteriormente.

Cuando el botón “Aceptar” es presionado, la función **`send()`** es llamada y el formulario creado dentro del objeto **`FormData`** es enviado al servidor. El archivo **`procesar.php`** recibe estos datos (**`nombre`** y **`apellido`**) y retorna un texto al navegador incluyendo esta información. La función **`mostrar()`** es ejecutada cuando el proceso es finalizado. La información recibida es mostrada en pantalla desde esta función a través de la propiedad **`responseText`**.

Hágalo usted mismo: Este ejemplo requiere que varios archivos sean subidos al servidor. Vamos a utilizar el mismo documento HTML y estilos CSS de los Listados 13-1 y 13-2. El código Javascript en el Listado 13-5 reemplaza al anterior. También debe crear un nuevo archivo llamado **`procesar.php`** con el código del

Listado 13-6. Suba todos estos archivos al servidor y abra el documento HTML en su navegador. Haciendo clic en el botón “Aceptar”, debería ver en pantalla el texto retornado por **procesar.php**.

Solicitudes de diferente origen

Hasta ahora hemos trabajado con códigos y archivos de datos ubicados en el mismo directorio y en el mismo dominio, pero XMLHttpRequest Level 2 nos deja hacer solicitudes a diferentes orígenes, lo que significa que podremos interactuar con diferentes servidores desde la misma aplicación.

El acceso de un origen a otro debe ser autorizado en el servidor. La autorización se realiza declarando los orígenes que tienen permiso para acceder a la aplicación. Esto es hecho en la cabecera enviada por el servidor que aloja el archivo que procesa la solicitud.

Por ejemplo, si nuestra aplicación está ubicada en www.dominio1.com y desde ella accedemos al archivo **procesar.php** ubicado en www.dominio2.com, el segundo servidor debe ser configurado para declarar al origen www.dominio1.com como un origen válido para una solicitud XMLHttpRequest.

Podemos especificar esta configuración desde los archivos de configuración del servidor, o declararlo en la cabecera desde el código. En el segundo caso, la solución para nuestro ejemplo sería tan simple como agregar la cabecera **Access-Control-Allow-Origin** al código del archivo **procesar.php**:

```
<?PHP
  header('Access-Control-Allow-Origin: *');

  print('Su nombre es: '.$_POST['nombre'].'<br>');
  print('Su apellido es: '.$_POST['apellido']);
?>
```

Listado 13-7. *Autorizando solicitudes de orígenes múltiples.*

El valor ***** para la cabecera **Access-Control-Allow-Origin** representa orígenes múltiples. El código del Listado 13-7 podrá ser accedido desde cualquier origen a menos que el valor ***** sea reemplazado por un origen específico (por ejemplo, **http://www.dominio1.com**, lo que solo autorizará a aplicaciones desde el dominio www.dominio1.com a acceder al archivo).

IMPORTANTE: El nuevo código PHP del Listado 13-7 agrega el valor solo a la cabecera retornada por el archivo **procesar.php**. Para incluir este parámetro en la cabecera de cada uno de los archivos retornados por el servidor, necesitamos modificar los archivos de configuración del servidor HTTP. Para encontrar más información al respecto, visite los enlaces correspondientes a este capítulo en nuestro sitio web o lea las instrucciones de su servidor HTTP.

Subiendo archivos

Subir archivos a un servidor es una tarea que tarde o temprano todo desarrollador debe enfrentar. Es una característica requerida por casi toda aplicación web estos días, pero no contemplada por navegadores hasta el momento. Esta API se hace cargo de la situación incorporando un nuevo atributo que retorna un objeto XMLHttpRequestUpload. Utilizando este objeto podemos acceder a todos los métodos, propiedades y eventos de un objeto XMLHttpRequest pero también controlar el proceso de subida.

upload Este atributo retorna un objeto XMLHttpRequestUpload. El atributo debe ser llamado desde un objeto XMLHttpRequest ya existente.

Para trabajar con este atributo vamos a necesitar una nueva plantilla con un elemento **<input>** desde el que seleccionaremos el archivo a ser subido:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
```

```
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Archivo a Subir:<br><input type="file" name="archivos"
                                              id="archivos"></p>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 13-8. *Plantilla para subir archivos.*

Para subir un archivo tenemos que usar una referencia al archivo y enviarla como un campo de formulario. El objeto **FormData** estudiado en el ejemplo anterior es capaz de manejar esta clase de datos. El navegador detecta automáticamente la clase de información agregada al objeto **FormData** y crea las cabeceras apropiadas para iniciar la solicitud. El resto del proceso es exactamente el mismo estudiado anteriormente.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  var archivos=document.getElementById('archivos');
  archivos.addEventListener('change', subir, false);
}
function subir(e){
  var archivos=e.target.files;
  var archivo=archivos[0];

  var datos=new FormData();
  datos.append('archivo',archivo);

  var url="procesar.php";
  var solicitud=new XMLHttpRequest();
  var xmlupload=solicitud.upload;
  xmlupload.addEventListener('loadstart',comenzar,false);
  xmlupload.addEventListener('progress',estado,false);
  xmlupload.addEventListener('load',mostrar,false);
  solicitud.open("POST", url, true);
  solicitud.send(datos);
}
function comenzar(){
  cajadatos.innerHTML='<progress value="0" max="100">0%</progress>';
}
function estado(e){
  if(e.lengthComputable){
    var por=parseInt(e.loaded/e.total*100);
    var barraprogreso=cajadatos.querySelector("progress");
    barraprogreso.value=por;
    barraprogreso.innerHTML=por+'%';
  }
}
function mostrar(e){
  cajadatos.innerHTML='Terminado';
}
window.addEventListener('load', iniciar, false);
```

Listado 13-9. *Subiendo un archivo con* `FormData`.

La principal función del Listado 13-9 es **subir()**. Esta función es llamada cuando el usuario selecciona un nuevo archivo desde el elemento **<input>** (y el evento **change** es disparado). El archivo seleccionado es recibido y almacenado en la variable **archivo**, exactamente del mismo modo que lo hicimos anteriormente para aplicar la API File en el Capítulo 12 y también para la API Drag and Drop en el Capítulo 8. Los métodos usados en cada ocasión retornan un objeto **File**.

Una vez que tenemos la referencia al archivo, el objeto **FormData** es creado y el archivo es agregado a este

objeto por medio del método **append()**. Para enviar este formulario, iniciamos una solicitud **POST**. Primero, un objeto XMLHttpRequest común es asignado a la variable de la solicitud. Más adelante, usando el atributo **upload**, un objeto XMLHttpRequestUpload es creado y almacenado en la variable **xmlupload**. Usando esta variable agregamos escuchas para todos los eventos disparados por el proceso de subida y finalmente la solicitud es enviada **send(datos)**.

El resto del código hace lo mismo que en el ejemplo del Listado 13-4; en otras palabras, una barra de progreso es mostrada en la pantalla cuando el proceso de subida comienza y luego es actualizada de acuerdo al progreso del mismo.

Hágalo usted mismo: En este ejemplo indicamos que el archivo **procesar.php** se encargará de procesar los archivos enviados al servidor, pero no hacemos nada al respecto. Para probar el código anterior, puede utilizar un archivo **procesar.php** vacío.

Aplicación de la vida real

Subir un archivo a la vez probablemente no sea lo que la mayoría de los desarrolladores tengan en mente. Así como tampoco lo es utilizar el elemento **<input>** para seleccionar los archivos a subir. En general, todo programador busca que sus aplicaciones sean lo más intuitivas posible, y qué mejor manera de lograrlo que combinando técnicas y métodos a los que los usuarios ya están familiarizados. Aprovechando API Drag and Drop, vamos a crear una aplicación que se asemeja a lo que normalmente usamos en la vida real. Los archivos podrán ser subidos al servidor simplemente arrastrándolos desde el Explorador de Archivos hasta un área en la página web.

Creemos primero un documento HTML con la caja donde soltar los archivos:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="cajadatos">
    <p>Suelte los archivos aquí</p>
  </section>
</body>
</html>
```

Listado 13-10. *Área para soltar los archivos a subir.*

El código Javascript para este ejemplo es probablemente el más complejo de los que hemos visto hasta el momento a lo largo del libro. No solo combina varias APIs sino también varias funciones anónimas para mantener todo organizado y dentro del mismo entorno (dentro de la misma función). Este código debe tomar los archivos soltados dentro del elemento **cajadatos**, listarlos en la pantalla, preparar el formulario virtual con esta información, hacer una solicitud para subir cada archivo al servidor y actualizar las barras de progreso de cada uno mientras son subidos.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  cajadatos.addEventListener('dragenter', function(e){
                                  e.preventDefault(); }, false);
  cajadatos.addEventListener('dragover', function(e){
                                  e.preventDefault(); }, false);
  cajadatos.addEventListener('drop', soltado, false);
}
function soltado(e){
  e.preventDefault();
  var archivos=e.dataTransfer.files;
  if(archivos.length){
    var lista='';
```

```
    for(var f=0;f<archivos.length;f++){
      var archivo=archivos[f];
      lista+='<blockquote>Archivo: '+archivo.name;
      lista+='<br><span><progress value="0" max="100">0%
                                       </progress></span>';
      lista+='</blockquote>';
    }
    cajadatos.innerHTML=lista;

    var cuenta=0;
    var subir=function(){
        var archivo=archivos[cuenta];
        var datos=new FormData();
        datos.append('archivo',archivo);
        var url="procesar.php";
        var solicitud=new XMLHttpRequest();
        var xmlupload=solicitud.upload;

        xmlupload.addEventListener('progress',function(e){
            if(e.lengthComputable){
              var hijo=cuenta+1;
              var por=parseInt(e.loaded/e.total*100);
              var barraprogreso=cajadatos.querySelector("block
                  quote:nth-child("+hijo+") > span > progress");
              barraprogreso.value=por;
              barraprogreso.innerHTML=por+'%';
            }
          },false);
        xmlupload.addEventListener('load',function(){
            var hijo=cuenta+1;
            var elemento=cajadatos.querySelector("blockquote:nth-
                                   child("+hijo+") > span");
            elemento.innerHTML='Terminado!';

            cuenta++;
            if(cuenta<archivos.length){
              subir();
            }
          },false);
        solicitud.open("POST", url, true);
        solicitud.send(datos);
      }
    subir();
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 13-11. *Subiendo archivos uno por uno.*

Bien, no es un código cómodo para analizar, pero será fácil de entender si lo estudiamos paso a paso. Como siempre, todo comienza con la llamada a la función **iniciar()** cuando el documento es completamente cargado. Esta función crea una referencia al elemento **cajadatos** que será la caja donde soltar los archivos, y agrega escuchas para tres eventos que controlan la operación de arrastrar y soltar. Para conocer cómo se procesa exactamente esta operación, lea nuevamente el Capítulo 8. Básicamente, el evento **dragenter** es disparado cuando los archivos que son arrastrados ingresan en el área del elemento **cajadatos**, el evento **dragover** es disparado periódicamente cuando los archivos arrastrados están sobre este elemento, y el evento **drop** es disparado cuando los archivos son finalmente soltados dentro de la caja en la pantalla. No debemos hacer nada para responder a los eventos **dragenter** y **dragover** en este ejemplo, por lo que los mismos son cancelados para prevenir el comportamiento por defecto del navegador. El único evento al que responderemos es **drop**. La escucha para este evento llama a la función **soltado()** cada vez que algo es soltado dentro de **cajadatos**.

La primera línea de la función **soltado()** también usa el método **preventDefault()** para poder hacer con los archivos lo que nosotros queremos y no lo que el navegador haría por defecto. Ahora que tenemos absoluto control de la situación, es tiempo de procesar los archivos soltados. Primero, necesitamos obtener la lista de archivos desde el elemento **dataTransfer**. El valor retornado es un array que almacenamos en la variable **archivos**. Para estar seguros de que lo que fue soltado son archivos y no otra clase de elementos, controlamos

el valor de la propiedad **length** con el condicional **if(archivos.length)**. Si este valor es diferente a 0 o **null** significa que uno o más archivos han sido soltados dentro de la caja y podemos continuar con el proceso.

Es hora de procesar los archivos recibidos. Con un bucle **for** navegamos a través del array **archivos** y creamos una lista de elementos **<blockquote>** conteniendo cada uno el nombre del archivo y una barra de progreso encerrada en etiquetas **<span>**. Una vez que la construcción de la lista es finalizada, el resultado es mostrado en pantalla como el contenido de **cajadatos**.

A simple vista parece que la función **soltado()** hace todo el trabajo, pero dentro de esta función creamos otra llamada **subir()** que se hace cargo del proceso de subir los archivos uno por uno al servidor. Por lo tanto, luego de mostrar todos los archivos en pantalla la siguiente tarea es crear esta función y llamarla por cada archivo en la lista.

La función **subir()** fue creada usando una función anónima. Dentro de esta función, primero seleccionamos un archivo desde el array usando la variable **cuenta** como índice. Esta variable fue previamente inicializada a **0**, por lo que la primera vez que la función **subir()** es llamada, el primer archivo de la lista es seleccionado y subido.

Para subir cada archivo usamos el mismo método que en anteriores ejemplos. Una referencia al archivo es almacenada en la variable **archivo**, un objeto **FormData** es creado usando el constructor **FormData()** y el archivo es agregado al objeto con el método **append()**.

En esta oportunidad, solo escuchamos a dos eventos: **progress** y **load**. Cada vez que el evento **progress** es disparado, una función anónima es llamada para actualizar el estado de la barra de progreso del archivo que está siendo subido. Para identificar el elemento **<progress>** correspondiente a ese archivo, usamos el método **querySelector()** con la pseudo clase **:nth-child()**. El índice de la pseudo clase es calculado usando el valor de la variable **cuenta**. Esta variable contiene el número del índice del array **archivos**, pero este índice comienza en **0** mientras que el índice para la lista de hijos accedidos por **:nth-child()** se inicia en el valor **1**. Para obtener el valor del índice correspondiente y referenciar el elemento **<progress>** correcto, agregamos **1** al valor de **cuenta**, almacenamos el resultado en la variable **hijo** y usamos esta variable como índice.

Cuando el proceso anterior es terminado, tenemos que informar sobre la situación y avanzar hacia el siguiente archivo en el array **archivos**. Para este propósito, en la función anónima ejecutada cuando el evento **load** es disparado, incrementamos el valor de **cuenta** en una unidad, reemplazamos el elemento **<progress>** correspondiente al archivo subido por el texto “Terminado!”, y llamamos a la función **subir()** nuevamente (siempre y cuando queden aún archivos por procesar).

Volvamos un poco a ver a grandes rasgos cómo el proceso es desarrollado. Si sigue el código del Listado 13-11, verá que, luego de declarar la función **subir()**, ésta es llamada por primera vez al final de la función **soltado()**. Debido a que la variable **cuenta** es inicializada a **0**, el primer archivo contenido en el array **archivos** será procesado en primer lugar. Más adelante, cuando este archivo es subido por completo, el evento **load** es disparado y la función anónima llamada para responder al mismo incrementará el valor de **cuenta** una unidad y ejecutará la función **subir()** nuevamente para procesar el archivo siguiente. Al final, todos los archivos arrastrados y soldados dentro de la caja en pantalla habrán sido subidos al servidor, uno por uno.

13.2 Cross Document Messaging

Esta parte de lo que llamamos API Communication es conocida oficialmente como API Web Messaging. Cross Document Messaging es una técnica que permite a aplicaciones de diferentes orígenes comunicarse entre sí. Aplicaciones funcionando en diferentes cuadros, ventanas o pestañas (o incluso otras APIs) pueden comunicarse ahora aprovechando esta tecnología. El procedimiento es simple: publicamos un mensaje desde un documento y lo procesamos en el documento destino.

Constructor

Para publicar mensajes, la API provee el método **`postMessage()`**:

postMessage(mensaje, destino) Este método es aplicado al **`contentWindow`** del objeto `Window` que recibe el mensaje. El atributo **`mensaje`** es una cadena de texto representando el mensaje a transmitir, y el atributo **`destino`** es el dominio del documento destino (que puede ser una URL o un puerto, como veremos más adelante). El destino puede ser declarado como un dominio específico, como cualquier documento usando el símbolo **`*`**, o como el mismo origen del documento que envía el mensaje usando el símbolo **`/`**. El método puede también incluir un array de puertos como tercer atributo.

Evento message y propiedades

El método de comunicación es asíncrono. Para escuchar por mensajes enviados por un documento en particular, la API ofrece el evento **`message`**, el cual incluye algunas propiedades con información sobre la operación:

data Esta propiedad retorna el contenido del mensaje.

origin Esta propiedad retorna el origen del documento que envió el mensaje, generalmente el dominio. Este valor puede ser usado luego para enviar un mensaje de regreso.

source Esta propiedad retorna un objeto usado para identificar a la fuente del mensaje. Este valor puede ser usado para apuntar al documento que envía el mensaje, como veremos más adelante.

Enviando mensajes

Para crear un ejemplo de esta API, tenemos que considerar lo siguiente: la comunicación ocurre entre diferentes ventanas (ventanas, cuadros, pestañas u otras APIs), debido a esto debemos generar documentos y códigos para cada extremo del proceso. Nuestro ejemplo incluye una plantilla con un iframe (cuadro interno) y los códigos Javascript apropiados para cada documento HTML. Comencemos por el documento principal:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Cross Document Messaging</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Su nombre: <input type="text" name="nombre" id="nombre"
                                                    required></p>
      <p><input type="button" name="boton" id="boton"
                                              value="Enviar"></p>
    </form>
  </section>
  <section id="cajadatos">
    <iframe id="iframe" src="iframe.html" width="500"
                                            height="350"></iframe>
  </section>
```

```
</body>
</html>
```

Listado 13-12. *Plantilla para comunicación entre documentos.*

Como podemos ver, al igual que en plantillas previas, incluimos dos elementos **`<section>`**, pero esta vez **`cajadatos`** contiene un **`<iframe>`** que cargará el archivo **`iframe.html`**. Vamos a volver a esto en un momento. Antes agreguemos algunos estilos a esta estructura:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 13-13. *Estilos para las cajas en pantalla (`messaging.css`).*

El código Javascript para el documento principal tiene que tomar el valor del campo **`nombre`** del formulario y enviarlo como un mensaje al documento dentro del iframe usando el método **`postMessage()`**:

```
function iniciar(){
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);
}
function enviar(){
  var nombre=document.getElementById('nombre').value;
  var iframe=document.getElementById('iframe');

  iframe.contentWindow.postMessage(nombre, '*');
}
window.addEventListener('load', iniciar, false);
```

Listado 13-14. *Publicando un mensaje (`messaging.js`).*

En el código del Listado 13-14, el mensaje fue compuesto por el valor del campo **`nombre`**. El símbolo **`*`** fue usado como valor de destino para enviar este mensaje a cualquier documento dentro del iframe (sin importar su origen).

Una vez que el botón “Enviar” es presionado, la función **`enviar()`** es llamada y el valor del campo es enviado al contenido del iframe. Ahora es momento de tomar ese mensaje y procesarlo. Recuerde que el documento destinado a ser abierto en un iframe es exactamente igual a uno abierto en la ventana principal. El iframe simplemente simula una ventana dentro del documento. Por este motivo, vamos a crear una pequeña plantilla similar a la anterior pero solo con el propósito de mostrar la información recibida:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>iframe window</title>
  <script src="iframe.js"></script>
</head>
<body>
  <section>
    <div><b>Mensaje desde la ventana principal:</b></div>
    <div id="cajadatos"></div>
  </section>
```

```
</body>
</html>
```

Listado 13-15. *Plantilla para el iframe (iframe.html).*

Esta plantilla tiene su propia **cajadatos** que será usada para mostrar el mensaje recibido en la pantalla, y también su propio código Javascript para procesarlo:

```
function iniciar(){
  window.addEventListener('message', receptor, false);
}
function receptor(e){
  var cajadatos=document.getElementById('cajadatos');
  cajadatos.innerHTML='mensaje desde: '+e.origin+'<br>';
  cajadatos.innerHTML+='mensaje: '+e.data;
}
window.addEventListener('load', iniciar, false);
```

Listado 13-16. *Procesando los mensajes desde el destino (iframe.js).*

Acorde a lo que explicamos anteriormente, para escuchar los mensajes la API provee el evento **message** y algunas propiedades. En el código del Listado 13-16, una escucha para este evento fue agregada y la función **receptor()** fue declarada para responder al mismo. Esta función muestra el contenido del mensaje usando la propiedad **data** e información acerca del documento que lo envió usando el valor de **origin**.

Recuerde que este código pertenece al documento HTML del iframe, no al documento principal del Listado 13-12. Estos son dos documentos diferentes con sus propios entornos, objetivos y códigos (uno es abierto en la ventana principal y el otro dentro del iframe).

> **Hágalo usted mismo:** Hay un total de cinco archivos que tienen que ser creados y subidos al servidor para poder probar este ejemplo. Primero, cree un nuevo archivo HTML con el código del Listado 13-12 que será nuestro documento principal. Este documento también requiere el archivo **messaging.css** con los estilos del Listado 13-13 y el archivo **messaging.js** con el código Javascript del Listado 13-14. La plantilla del Listado 13-12 contiene un elemento **<iframe>** con el archivo **iframe.html** como su fuente. Necesitará crear también este archivo y copiar en su interior el código del Listado 13-15 y además generar el correspondiente archivo **iframe.js** con los códigos del Listado 13-16. Suba todos los archivos a su servidor, abra el primer documento HTML en su navegador y envíe su nombre o cualquier texto al iframe usando el formulario en pantalla.

Filtros y múltiples orígenes

Lo que hemos hecho hasta ahora no es una práctica recomendable, especialmente si consideramos asuntos de seguridad. El código en el documento principal envía un mensaje a un iframe específico, pero no controla los documentos dentro de ese iframe que estarán autorizados a leerlo (cualquier documento dentro del iframe podrá leer el mensaje). Del mismo modo, el código de nuestro ejemplo para el iframe no controla el origen y procesa todo mensaje recibido. Ambas partes del proceso de comunicación tienen que ser mejoradas para prevenir abusos o errores.

En el siguiente ejemplo, vamos a corregir esta situación y estudiar el procedimiento a seguir para responder a un mensaje del documento origen usando otra propiedad del evento **message** llamada **source**.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Cross Document Messaging</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Su nombre: <input type="text" name="nombre" id="nombre"
```

```
                                                          required></p>
        <p><input type="button" name="boton" id="boton"
                                                    value="Enviar"></p>
      </form>
    </section>
    <section id="cajadatos">
      <iframe id="iframe" src="http://www.dominio2.com/iframe.html"
                                width="500" height="350"></iframe>
    </section>
  </body>
  </html>
```

Listado 13-17. *Comunicándonos con un origen/destino específicos.*

Supongamos que el nuevo documento HTML con el código del Listado 13-17 está localizado en www.dominio1.com, pero el código para el iframe es cargado desde una segunda ubicación en www.dominio2.com. Para prevenir abusos y errores, necesitaremos declarar estos dominios en el código y ser específicos sobre quién estará autorizado a leer los mensajes y desde dónde.

En el código del Listado 13-17, no estamos solo cargando el archivo HTML para el iframe sino declarando la ruta completa hacia otro servidor (www.dominio2.com). El documento principal se encontrará en www.dominio1.com y el contenido del iframe en www.dominio2.com. Los siguientes códigos consideran esta situación:

```
function iniciar(){
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  window.addEventListener('message', receptor, false);
}
function enviar(){
  var nombre=document.getElementById('nombre').value;
  var iframe=document.getElementById('iframe');
  iframe.contentWindow.postMessage(nombre, 'http://www.dominio2.com');
}
function receptor(e){
  if(e.origin=='http://www.dominio2.com'){
    document.getElementById('nombre').value=e.data;
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 13-18. *Comunicándonos con orígenes específicos (`messaging.js`).*

Preste atención a la función **`enviar()`** en el Listado 13-18. El método **`postMessage()`** ahora declara un destino específico para el mensaje (www.dominio2.com). Solo documentos dentro del iframe y que provengan de ese origen específico podrán leer este mensaje.

En la función **`iniciar()`** del Listado 13-18 también agregamos una escucha para el evento **`message`**. El propósito de esta escucha y de la función **`receptor()`** es recibir la respuesta enviada desde el iframe. Esto cobrará sentido en unos minutos.

Veamos ahora el código Javascript ejecutado en el iframe que nos ayudará a entender cómo un mensaje proveniente de un origen específico es procesado y cómo respondemos al mismo (usaremos exactamente el mismo documento HTML del Listado 13-15 para el iframe).

```
function iniciar(){
  window.addEventListener('message', receptor, false);
}
function receptor(e){
  var cajadatos=document.getElementById('cajadatos');
  if(e.origin=='http://www.dominio1.com'){
    cajadatos.innerHTML='mensaje válido: '+e.data;
    e.source.postMessage('mensaje recibido', e.origin);
  }else{
```

```
      cajadatos.innerHTML='origen inválido';
    }
  }
  window.addEventListener('load', iniciar, false);
```

Listado 13-19. *Respondiendo al documento principal (`iframe.js`).*

Filtrar el origen es tan simple como comparar el valor de la propiedad **`origin`** con el dominio del cual queremos leer los mensajes. Una vez que comprobamos que el origen es válido, el mensaje es mostrado en pantalla y luego una respuesta es enviada de regreso aprovechando el valor de la propiedad **`source`**. La propiedad **`origin`** es también usada para declarar que esta respuesta estará solo disponible para la ventana que envió el mensaje en primer lugar. Ahora puede regresar al Listado 13-18 para comprender cómo la función **`receptor()`** procesará esta respuesta.

> **Hágalo usted mismo:** Este último ejemplo es un poco engañoso. Estamos usando dos orígenes diferentes, por lo que necesitará dos dominios diferentes (o subdominios) para comprobar el funcionamiento de los códigos. Reemplace los dominios declarados en los códigos por los suyos propios y luego suba los archivos correspondientes al documento principal en uno y los correspondientes al iframe en el otro. El documento principal cargará en el iframe el código desde el segundo dominio y así podrá ver cómo funciona el proceso de comunicación entre estos dos orígenes diferentes.

13.3 Web Sockets

En esta parte del capítulo, describiremos el último componente de lo que consideramos API Communication. API WebSocket ofrece soporte para comunicaciones bidireccionales entre navegadores y servidores. La comunicación es realizada a través de conexiones TCP, sin enviar cabeceras HTTP, reduciendo de este modo la cantidad de datos transmitidos en cada llamada. Además, la conexión es persistente, permitiendo a los servidores mantener a los navegadores permanentemente informados. Esto significa que no deberemos encargarnos de llamar al servidor a cada momento para obtener datos actualizados; en su lugar, el servidor mismo de forma automática nos enviará información acerca de la situación actual.

WebSocket puede ser considerado por algunos como una mejora de Ajax, pero es en realidad una alternativa totalmente diferente de comunicación que permite la construcción de aplicaciones en tiempo real en una plataforma escalable (por ejemplo, video juegos para múltiples jugadores, salas de chat, etc…).

La API es simple. Solo unos pocos métodos y eventos son incluidos para abrir y cerrar conexiones y enviar y escuchar por mensajes. Sin embargo, ningún servidor soporta este protocolo por defecto. Debido a esto, necesitaremos instalar nuestro propio servidor WS (servidor WebSocket) para poder establecer comunicación entre el navegador y el servidor que aloja a la aplicación.

Configuración del servidor WS

Un programador experimentado seguramente podrá descubrir por sí mismo cómo construir un servidor WS, pero para aquellos que deseamos dedicar nuestro tiempo libre a actividades un tanto más recreativas, ya se encuentran disponibles en la web varios códigos que nos permitirán configurar nuestro propio servidor WS y procesar conexiones en unos pocos minutos. Dependiendo de sus preferencias, puede optar por códigos programados en PHP, Java, Ruby, y otros.

> **IMPORTANTE:** Al momento de escribir estas líneas, la especificación está siendo mejorada y expandida debido a problemas de seguridad y aún no se encuentran servidores WS disponibles que soporten estas mejoras. Para obtener una lista completa, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Los siguientes ejemplos estarán orientados al uso de un servidor sencillo programado en PHP llamado phpwebsocket (code.google.com/p/phpwebsocket/). Este servidor utiliza un único archivo llamado **`server.php`** que responde a una serie de códigos pre programados, como veremos más adelante. El archivo debe ser subido a un servidor y luego ejecutado por medio de un sistema del tipo Telnet como Putty, por ejemplo.

WebSocket usa una conexión persistente, por lo que el código del servidor WS tiene que funcionar todo el tiempo, capturando solicitudes y enviando actualizaciones a los navegadores conectados. Usando Putty puede acceder a su servidor desde una consola y ejecutar los comandos necesarios para poner el servidor WS en marcha.

> **Hágalo usted mismo:** En primer lugar debe contar con las herramientas adecuadas. Instale su consola de acceso Telnet o descargue Putty desde www.chiark.
> greenend.org.uk/~sgtatham/putty/. También necesita descargar el archivo **server.php** del servidor phpwebsocket disponible en http://code.google.com/
> p/phpwebsocket/. Modifique los datos de acceso dentro de este archivo (dominio o IP de su servidor y puerto), y súbalo a su servidor. Desde la consola Telnet acceda al servidor y ejecute el archivo con el siguiente comando: **php –q server.php**. Esto pondrá en marcha el servidor WS.

> **IMPORTANTE:** El servidor no solo se encarga de establecer la comunicación entre el navegador y el servidor sino que además está a cargo de generar la respuesta adecuada. La forma de construir y realizar esta respuesta está programada dentro del mismo código del servidor. Deberá adaptar este código a las necesidades de su aplicación.

Constructor

Antes de programar los códigos para interactuar con el servidor WS, veamos lo que la API ofrece con este fin. La especificación declara solo una interface con unos pocos métodos, propiedades y eventos, además de un constructor, para establecer la conexión:

WebSocket(url) Este constructor inicia una conexión entre la aplicación y el servidor WS apuntado por el atributo **`url`**. Retorna un objeto WebSocket referenciando esta conexión. Un segundo atributo puede ser especificado para proveer un array con sub protocolos de comunicación.

Métodos

La conexión es iniciada por el constructor, por lo que solo necesitamos dos métodos para utilizarla:

send(datos) Este es el método necesario para enviar un mensaje al servidor WS. El valor del atributo `datos` representa los datos a ser transmitidos (normalmente una cadena de texto).

close() Este método cierra la conexión.

Propiedades

Algunas propiedades no permiten conocer la configuración y el estado de la conexión:

url Muestra la URL a la cual la aplicación está conectada.

protocol Esta propiedad retorna el sub protocolo usado, si existe.

readyState Esta propiedad retorna un número representando el estado de la conexión: 0 significa que la conexión no ha sido aún establecida, 1 significa que la conexión fue abierta, 2 significa que la conexión está siendo cerrada, y 3 significa que la conexión fue cerrada.

bufferedAmount Esta es una propiedad extremadamente útil que nos permite conocer la cantidad de datos requeridos pero aún no enviados al servidor. El valor retornado nos ayuda a regular la cantidad de datos y la frecuencia de cada solicitud para evitar saturar al servidor.

Eventos

Para conocer el estado de la conexión y escuchar por mensajes enviados por el servidor, debemos usar eventos. La API ofrece los siguientes:

open Este evento es disparado cuando la conexión es abierta.

message Este evento es disparado cuando un mensaje proveniente del servidor se encuentra disponible.

error Este evento es disparado cuando ocurre un error.

close Este evento es disparado cuando la conexión es cerrada.

Plantilla

El archivo **`server.php`** del servidor WS que usamos como ejemplo contiene una función **`process()`** que procesa una pequeña lista de comandos predefinidos y envía de regreso la respuesta apropiada. Para probar esta API, vamos a usar un formulario en el que podremos ingresar uno de estos comandos y enviarlos al servidor:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>WebSocket</title>
  <link rel="stylesheet" href="websocket.css">
  <script src="websocket.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Comando:<br><input type="text" name="comando"
                                          id="comando"></p>
      <p><input type="button" name="boton" id="boton"
                                        value="Enviar"></p>
    </form>
  </section>
  <section id="cajadatos"></section>
```

```
</body>
</html>
```

Listado 13-20. *Insertando commandos.*

También crearemos un archivo CSS llamado **websocket.css** con los siguientes estilos:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  height: 350px;
  overflow: auto;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 13-21. *Estilos habituales para las cajas.*

Iniciar la comunicación

Como siempre, el código Javascript es responsable de todo el proceso. En el siguiente listado, crearemos nuestra primera aplicación de comunicaciones para entender la forma de trabajo de esta API:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  socket=new WebSocket("ws://www.dominio.com:12345/server.php");
  socket.addEventListener('message', recibido, false);
}
function recibido(e){
  var lista=cajadatos.innerHTML;
  cajadatos.innerHTML='Recibido: '+e.data+'<br>'+lista;
}
function enviar(){
  var comando=document.getElementById('comando').value;
  socket.send(comando);
}
window.addEventListener('load', iniciar, false);
```

Listado 13-22. *Enviando mensajes al servidor.*

En la función **iniciar()**, el objeto WebSocket es construido y almacenado en la variable **socket**. El atributo **url** del constructor apunta hacia la ubicación del archivo **server.php** en nuestro servidor, incluyendo el puerto de conexión (en este ejemplo, 12345). Generalmente la dirección del servidor WS será declarada con el valor IP correspondiente al servidor (en lugar de un dominio) y un valor de puerto como 8000 u 8080, pero esto dependerá de sus necesidades, la configuración de su servidor, la ubicación de su archivo, etc… El uso de la IP en lugar del dominio es una práctica recomendada para evitar el proceso de traducción DNS. En cada una de las llamadas, la red realiza un proceso de traducción de las direcciones web para obtener las direcciones reales de los servidores a los que corresponden. Si en lugar de especificar un dominio declaramos directamente la dirección IP de nuestro servidor, evitamos esta operación, estableciendo una comunicación más fluida.

Luego de que obtenemos el objeto WebSocket, una escucha para el evento **message** es agregada. Este evento será disparado cada vez que el servidor WS envíe un mensaje al navegador. La función **recibido()** fue

declarada para responder al mismo. Como en otras API, este evento también incluye la propiedad **data** que retorna el contenido del mensaje. En la función **recibido()**, usamos esta propiedad para mostrar el mensaje en pantalla.

Para enviar mensajes al servidor incluimos la función **enviar()**. El valor insertado en el elemento **<input>** llamado **comando** es tomado por esta función y enviado al servidor WS usando el método **send()**.

> **IMPORTANTE:** El archivo **server.php** contiene una función llamada **process()** para procesar cada llamada y enviar una respuesta acorde a los datos recibidos. Usted puede cambiar esta función para satisfacer las necesidades de su aplicación, pero para nuestros ejemplos la hemos considerado exactamente como es ofrecido en Google Codes. La función toma el valor del mensaje recibido y lo compara con una lista de comandos predefinidos. Los comandos disponibles en la versión que utilizamos para probar estos ejemplos son: **hello**, **hi**, **name**, **age**, **date**, **time**, **thanks**, y **bye**. Por ejemplo, si enviamos el mensaje "hello", el servidor nos responderá "hello human".

Aplicación completa

En nuestro primer ejemplo podemos ver cómo funciona el proceso de comunicación de esta API. El constructor WebSocket inicia la conexión, el método **send()** envía cada mensaje al servidor para ser procesado, y el evento **message** informa a la aplicación sobre las respuestas recibidas. Sin embargo, no cerramos la conexión, no controlamos por errores, e incluso no detectamos cuándo la conexión estaba lista para trabajar. Veamos ahora un ejemplo que aprovecha todos los eventos provistos por la API para informar sobre el estado de la conexión en cada paso del proceso.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  socket=new WebSocket("ws://www.dominio.com:12345/server.php");
  socket.addEventListener('open', abierto, false);
  socket.addEventListener('message', recibido, false);
  socket.addEventListener('close', cerrado, false);
  socket.addEventListener('error', errores, false);
}
function abierto(){
  cajadatos.innerHTML='CONEXION ABIERTA<br>';
  cajadatos.innerHTML+='Estado: '+socket.readyState;
}
function recibido(e){
  var lista=cajadatos.innerHTML;
  cajadatos.innerHTML='Recibido: '+e.data+'<br>'+lista;
}
function cerrado(){
  var lista=cajadatos.innerHTML;
  cajadatos.innerHTML='CONEXION CERRADA<br>'+lista;

  var boton=document.getElementById('boton');
  boton.disabled=true;
}
function errores(){
  var lista=cajadatos.innerHTML;
  cajadatos.innerHTML='ERROR<br>'+lista;
}
function enviar(){
  var comando=document.getElementById('comando').value;
  if(comando=='cerrar'){
    socket.close();
  }else{
    socket.send(comando);
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 13-23. *Informando el estado de la conexión.*

Incluimos algunas mejoras en el código del Listado 13-23 comparado con el ejemplo anterior. Escuchas para todos los eventos disponibles en el objeto WebSocket fueron agregadas junto con las funciones apropiadas para responder a cada uno de ellos. También mostramos el estado de la conexión cuando es abierta usando el valor de la propiedad **`readyState`**, cerramos la conexión usando el método **`close()`** cuando el comando "cerrar" es enviado por el usuario desde el formulario, y desactivamos el botón "Enviar" cuando la conexión es cerrada (**`boton.disabled=true`**).

> **Hágalo usted mismo:** Este último ejemplo requiere el documento HTML y los estilos CSS de los Listados 13-20 y 13-21. Suba estos archivos a su servidor y, si aún no lo ha hecho, ejecute el servidor WS con la instrucción **`php -q server.php`** (como ya explicamos, esto es realizado desde una consola Telnet como Putty). Una vez que el servidor es activado, abra el documento HTML en su navegador. Inserte comandos en el formulario en pantalla y presione el botón "Enviar". Debería obtener respuestas del servidor de acuerdo al comando insertado (**`hello`**, **`hi`**, **`name`**, **`age`**, **`date`**, **`time`**, **`thanks`**, o **`bye`**). Envíe el comando "cerrar" para cerrar la conexión.

> **IMPORTANTE:** Si el servidor no funciona correctamente, los errores producidos serán retornados dentro de la consola Telnet. Controle esta consola para descubrir inconvenientes en la conexión.

13.4 Referencia rápida

HTML5 incorpora tres API diferentes con propósitos de comunicación. XMLHttpRequest Level 2 es una mejora del viejo objeto XMLHttpRequest usado para la construcción de aplicaciones Ajax. La API Web Messaging ofrece un sistema de comunicación para diferentes ventanas, pestañas, iframes, o incluso otras APIs. Y la API WebSocket provee una nueva alternativa para establecer una conexión rápida y efectiva entre navegadores y servidores.

XMLHttpRequest Level 2

Esta API tiene un constructor para objetos XMLHttpRequest y algunos métodos, propiedades y eventos para procesar la conexión.

XMLHttpRequest() Este constructor retorna el objeto XMLHttpRequest que necesitamos para iniciar y procesar una conexión con el servidor.

open(método, url, asíncrono) Este método abre la conexión entre la aplicación y el servidor. El atributo `método` determina qué método HTTP será usado para enviar la información (`GET` o `POST`). El atributo `url` declara la ruta hacia el código que recibirá y procesará esta información. Y el atributo `asíncrono` es un valor booleano que establece si la conexión será síncrona o asíncrona (`true` para asíncrona).

send(datos) Este método envía el valor del atributo `datos` al servidor. El atributo `datos` puede ser un ArrayBuffer, un blob, un documento, una cadena de texto o un objeto FormData.

abort() Este método cancela la solicitud.

timeout Esta propiedad establece el tiempo en milisegundos que tiene la solicitud para ser procesada.

readyState Esta propiedad retorna un valor representando el estado de la conexión: 0 significa que el objeto fue construido, 1 significa que la conexión fue abierta, 2 significa que la cabecera de la respuesta ha sido recibida, 3 significa que la respuesta está siendo recibida, 4 significa que la transmisión de datos ha sido completada.

responseType Esta propiedad retorna el tipo de respuesta. Puede ser declarada para cambiar el tipo de respuesta. Los posibles valores son `arraybuffer`, `blob`, `document`, y `text`.

response Esta propiedad retorna la respuesta a la solicitud en el formato declarado por la propiedad `responseType`.

responseText Esta propiedad retorna la respuesta a la solicitud en formato texto.

responseXML Esta propiedad retorna la respuesta a la solicitud como si fuera un documento XML.

loadstart Este evento es disparado cuando la solicitud es iniciada.

progress Este evento es disparado periódicamente mientras la solicitud es procesada.

abort Este evento es disparado cuando la solicitud es abortada.

error Este evento es disparado cuando ocurre un error.

load Este evento es disparado cuando la solicitud ha sido completada exitosamente.

timeout Este evento es disparado cuando la solicitud toma más tiempo en ser procesada que el especificado por la propiedad `timeout`.

loadend Este evento es disparado cuando la solicitud ha sido completada (en ambos casos, éxito o error).

Un atributo especial fue incluido para obtener un objeto XMLHttpRequestUpload en lugar del objeto XMLHttpRequest con el propósito de subir datos al servidor.

upload Este atributo retorna un objeto XMLHttpRequestUpload. Este objeto usa los mismos métodos, propiedades y eventos de un objeto XMLHttpRequest pero con el propósito de procesar la subida de archivos.

La API también incluye una interface para crear objetos FormData representando formularios HTML.

FormData() Este constructor retorna un objeto FormData para representar un formulario HTML.

append(nombre, valor) Este método agrega datos a un objeto FormData. Cada dato agregado al objeto representa un campo de formulario, con su nombre y valor declarado en los atributos. Una cadena de texto o un blob pueden ser provistos para el atributo `valor`.

Esta API usa la interface ProgressEvent (también usada por otras APIs para controlar el progreso de una operación) que incluye las siguientes propiedades:

lengthComputable Esta propiedad es un valor booleano que determina si los valores del resto de las propiedades son válidos.

loaded Esta propiedad retorna la cantidad total de bytes ya descargados o subidos.

total Esta propiedad retorna el tamaño total en bytes de los datos que están siendo descargados o subidos.

API Web Messaging

Esta API está constituida solo por una interface que provee métodos, propiedades y eventos para comunicar entre sí aplicaciones ubicadas en diferentes ventanas, pestañas, iframes o incluso otras API.

postMessage(mensaje, destino) Este método envía un mensaje a una `contentWindow` específica y al documento declarado como destino por el atributo `destino`. El atributo `mensaje` es el mensaje a ser transmitido.

message Este evento es disparado cuando un mensaje es recibido.

data Esta propiedad del evento `message` retorna el contenido del mensaje recibido.

origin Esta propiedad del evento `message` retorna el origen del documento que envió el mensaje.

source Esta propiedad del evento `message` retorna una referencia a la ventana desde la que el mensaje fue enviado.

API WebSocket

Esta API incluye un constructor que retorna un objeto WebSocket e inicia la conexión. Además, provee algunos métodos, propiedades y eventos para controlar la comunicación entre el navegador y el servidor.

WebSocket(url) Este constructor retorna un objeto WebSocket e inicia la conexión con el servidor. El atributo `url` declara la ruta del código del servidor WS y el puerto de comunicación. Un array con sub protocolos puede ser especificado como un segundo atributo.

send(datos) Este método envía un mensaje al servidor WS. El atributo `datos` debe ser una cadena de texto con el mensaje a ser enviado.

close() Este método cierra la conexión con el servidor WS.

url Esta propiedad muestra la URL que la aplicación está usando para conectarse al servidor WS.

protocol Esta propiedad retorna el sub protocolo usado por la conexión, si existe.

readyState Esta propiedad retorna un valor representando el estado de la conexión: 0 significa que la conexión no ha sido aún establecida, 1 significa que la conexión fue abierta, 2 significa que la conexión está siendo cerrada, y 3 significa que la conexión fue cerrada.

bufferedAmount Esta propiedad retorna la cantidad total de datos que esperan ser enviados al servidor.

open Este evento es disparado cuando la conexión es abierta.

message Este evento es disparado cuando el servidor envía un mensaje a la aplicación.

error Este evento es disparado cuando ocurre un error.

close Este evento es disparado cuando la conexión es cerrada.

Capítulo 14
API Web Workers

14.1 Haciendo el trabajo duro

Javascript se ha convertido en la principal herramienta para la construcción de aplicaciones exitosas en Internet. Como explicamos en el Capítulo 4, ya no es solo una alternativa para la creación de llamativos (a veces irritantes) trucos para la web. El lenguaje se ha vuelto una parte esencial de la web y una tecnología que cada desarrollador necesita entender e implementar.

Javascript ya ha alcanzado el estado de lenguaje de propósito general, una condición en la cual es forzado a proveer características elementales que por naturaleza no posee. Este lenguaje fue concebido como un lenguaje interpretado, creado con la intención de ser procesado un código a la vez. La ausencia de multiprocesamiento en Javascript (procesamiento de múltiples códigos al mismo tiempo) reduce su eficiencia, limita su alcance y vuelve a algunas aplicaciones de escritorio imposibles de emular en la web.

Web Workers es una API diseñada con el propósito específico de convertir Javascript en un lenguaje multiproceso y resolver este problema. Ahora, gracias a HTML5, podemos ejecutar códigos exigentes detrás de escena mientras el código principal sigue siendo ejecutado en la página web.

Creando un trabajador

La forma en la que Web Workers trabaja es simple: el trabajador (worker) es construido en un archivo Javascript separado y los códigos son comunicados entre sí a través de mensajes. Normalmente, el mensaje enviado al trabajador (worker) desde el código principal es la información que queremos que sea procesada, mientras que los mensajes enviados en respuesta desde el trabajador representan el resultado de este procesamiento. Para enviar y recibir estos mensajes, la API aprovecha técnicas implementadas en otras API ya estudiadas. Eventos y métodos que ya conocemos son usados para enviar y recibir mensajes desde un código al otro. Los siguientes son los elementos provistos por la API con este propósito:

Worker(códigoURL) Antes de comunicarnos con el trabajador, debemos obtener un objeto que apunta al archivo que contiene el código del trabajador. Este método retorna un objeto Worker. El atributo **`códigoURL`** es la URL del archivo que contiene el código que será ejecutado detrás de escena.

postMessage(mensaje) Este método es el mismo estudiado antes en el Capítulo 13 para Web Messaging API, pero ahora implementado para el objeto Worker. El método envía un mensaje hacia o desde el código del trabajador. El atributo **`mensaje`** es una cadena de texto o un objeto JSON representando el mensaje a ser transmitido.

message Este es un evento ya estudiado que escucha por mensajes enviados al código. Del mismo modo que el método **`postMessage()`**, este evento puede ser aplicado en el código principal o en el trabajador. Utiliza la propiedad **`data`** para obtener el mensaje enviado.

Enviando y recibiendo mensajes

Para estudiar cómo los trabajadores y el código principal se comunican entre sí, vamos a usar una plantilla simple conteniendo un formulario donde ingresar nuestro nombre y una caja donde mostrar la respuesta recibida.

Todo ejemplo de Web Workers, incluso el más sencillo, requiere al menos tres archivos: el documento principal, el código Javascript principal, y el archivo con el código para el trabajador. El siguiente será nuestro documento HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>WebWorkers</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
```

```
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Nombre:<br><input type="text" name="nombre"
                                          id="nombre"></p>
      <p><input type="button" name="boton" id="boton"
                                       value="Enviar"></p>
    </form>
  </section>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 14-1. *Plantilla para probar Web Workers.*

En la plantilla incluimos un archivo CSS llamado **`webworkers.css`** que contendrá las siguientes reglas:

```
#cajaformulario{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Listado 14-2. *Estilos para las cajas.*

El código Javascript para el documento principal tiene que ser capaz de enviar la información que queremos procesar en el trabajador. También debe estar preparado para escuchar por respuestas.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  trabajador=new Worker('trabajador.js');
  trabajador.addEventListener('message', recibido, false);
}
function enviar(){
  var nombre=document.getElementById('nombre').value;
  trabajador.postMessage(nombre);
}
function recibido(e){
  cajadatos.innerHTML=e.data;
}
window.addEventListener('load', iniciar, false);
```

Listado 14-3. *Un uso simple de la API.*

El Listado 14-3 presenta el código para nuestro documento principal (el que se encuentra dentro del archivo **`webworkers.js`**). En la función **`iniciar()`**, luego de la creación de las necesarias referencias para el elemento **`cajadatos`** y el botón del formulario, el objeto Worker es construido. El constructor **`Worker()`** declara al archivo **`trabajador.js`** como el contenedor del código para el trabajador y retorna un objeto Worker referenciando a este archivo. Todo proceso de interacción con este objeto será en realidad una interacción con el código de ese archivo.

Luego de la construcción de este objeto, agregamos una escucha para el evento **`message`** con la intensión de escuchar por mensajes provenientes del trabajador. Cuando un mensaje es recibido, la función **`recibido()`** es

llamada y el valor de la propiedad **data** (el mensaje) es mostrado en pantalla.

La otra parte de la comunicación es realizada por la función **enviar()**. Cuando el usuario hace clic sobre el botón "Enviar", el valor del campo **nombre** es enviado como un mensaje hacia el trabajador usando **postMessage()**.

Con las funciones **recibido()** y **enviar()** a cargo de la comunicación, estamos listos para enviar mensajes al trabajador y procesar sus respuestas. Es momento de preparar el código para el trabajador:

```
addEventListener('message', recibido, false);

function recibido(e){
  var respuesta='Su nombre es '+e.data;
  postMessage(respuesta);
}
```

Listado 14-4. *Código para el trabajador (trabajador.js).*

Del mismo modo que el código principal, el código para el trabajador tiene que escuchar constantemente por mensajes usando el evento **message**. La primera línea del código en el Listado 14-4 agrega una escucha para este evento. Esta escucha ejecutará la función **recibido()** cada vez que el evento es disparado (se recibe un mensaje desde el código principal). En esta función, el valor de la propiedad **data** es agregado a una cadena de texto predefinida y el resultado es enviado como respuesta usando nuevamente el método **postMessage()**.

Hágalo usted m,ismo: Compare los códigos de los Listados 14-3 y 14-4 (el código principal y el trabajador). Estudie cómo trabaja el procedimiento de comunicación y cómo los mismos métodos y eventos son aplicados con este propósito en cada uno de los códigos. Cree los archivos necesarios para probar este ejemplo usando los Listados 14-1, 14-2, 14-3 y 14-4, súbalos a su servidor y abra el documento HTML principal en su navegador.

IMPORTANTE: Puede usar los prefijos **self** o **this** para referenciar el trabajador (por ejemplo, **self.postMessage()**), o simplemente declare los métodos del modo que lo hicimos en el Listado 14-4.

Este trabajador es, por supuesto, extremadamente elemental. Nada es realmente procesado. El único proceso realizado es la construcción de una cadena de texto con el mensaje recibido y el envío de la misma de regreso como respuesta. Sin embargo, este ejemplo es útil para entender cómo los códigos se comunican entre sí y cómo podemos aprovechar esta API.

A pesar de su simplicidad, existen algunas cosas importantes que deberemos considerar antes de crear nuestros trabajadores. Los mensajes son la única forma de comunicarse directamente con los trabajadores. Estos mensajes, además, tienen que ser creados usando cadenas de texto u objetos JSON debido a que los trabajadores no pueden recibir otro tipo de datos. Tampoco pueden acceder al documento, manipular elementos HTML, y no tienen acceso a funciones y variables del código principal. Los trabajadores son como códigos enlatados, solo pueden acceder a información recibida a través de mensajes y enviar los resultados usando el mismo mecanismo.

Detectando errores

A pesar de las limitaciones mencionadas, los trabajadores son flexibles y poderosos. Podemos usar funciones, métodos Javascript predefinidos y otras API desde el interior de un trabajador. Considerando la complejidad que estos códigos pueden alcanzar, la API Web Workers incorpora un evento específico para informar sobre errores y retornar toda la información posible acerca de la situación.

error Este evento es disparado por el objeto Worker en el código principal cada vez que ocurre un error en el trabajador. Usa tres propiedades para retornar información: **message**, **filename** y **lineno**. La propiedad **message** retorna el mensaje de error. Es una cadena de texto que nos informa que salió mal. La propiedad **filename** muestra el nombre del archivo con el código que causó el error. Esto es útil cuando archivos externos son cargados desde el trabajador, como veremos más adelante. Finalmente, la propiedad **lineno** retorna el número de línea en la cual el error ocurrió.

Veamos un ejemplo de un código que muestra errores generados por el trabajador:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
```

```
    var boton=document.getElementById('boton');
    boton.addEventListener('click', enviar, false);

    trabajador=new Worker('trabajador.js');
    trabajador.addEventListener('error', errores, false);
  }
  function enviar(){
    var nombre=document.getElementById('nombre').value;
    trabajador.postMessage(nombre);
  }
  function errores(e){
    cajadatos.innerHTML='ERROR: '+e.message+'<br>';
    cajadatos.innerHTML+='Archivo: '+e.filename+'<br>';
    cajadatos.innerHTML+='Línea: '+e.lineno;
  }
  window.addEventListener('load', iniciar, false);
```

Listado 14-5. *Usando el evento* `error`.

El último código presentado es similar al código principal del Listado 14-3. Construye el trabajador pero solo utiliza el evento **`error`** debido a que no queremos escuchar por mensajes desde el trabajador en este ejemplo, solo controlar errores. Es inútil, por supuesto, pero nos mostrará cómo los errores son retornados y qué clase de información es ofrecida en estas situaciones.

Para generar un error deliberadamente podemos llamar a una función no existente dentro del trabajador, como muestra el siguiente ejemplo:

```
addEventListener('message', recibido, false);

function recibido(e){
  prueba();
}
```

Listado 14-6. *Un trabajador que no trabaja.*

En el trabajador debemos usar el evento **`message`** para escuchar por mensajes provenientes del código principal, al igual que hicimos anteriormente, porque ésta es la forma en la que el proceso comienza. Cuando un mensaje es recibido, la función **`recibido()`** es ejecutada y la función no existente **`prueba()`** es llamada, generando de este modo un error.

Tan pronto como el error ocurre, el evento **`error`** es disparado en el código principal y la función **`errores()`** es llamada, mostrando en pantalla los valores de las tres propiedades provistas por el evento. Estudie el código del Listado 14-5 para entender cómo la función toma y procesa esta información.

> **Hágalo usted mismo:** Para este ejemplo, usamos el documento HTML y las reglas CSS de los Listados 14-1 y 14-2. Copie el código del Listado 14-5 en el archivo **`webworkers.js`** y el código del Listado 14-6 en el archivo **`trabajador.js`**. Abra la plantilla del Listado 14-1 en su navegador y envié desde el formulario cualquier texto al trabajador. El error retornado por el trabajador será mostrado en pantalla.

Deteniendo trabajadores

Los trabajadores son unidades especiales de código que están constantemente trabajando detrás de escena, esperando por información para ser procesada. Los trabajadores serán, la mayoría de las veces, solo requeridos en circunstancias específicas y para propósitos especiales. Normalmente sus servicios no serán necesarios o requeridos todo el tiempo, por lo que será una buena práctica detenerlos o terminar sus procesos si ya no los necesitamos.

Con este objetivo, la API provee dos métodos diferentes:

terminate() Este método detiene el trabajador desde el código principal.

close() Este método detiene el trabajador desde dentro del trabajador mismo.

Cuando un trabajador es detenido, todo proceso que aún se encuentra desarrollando es abortado y toda tarea en el bucle de eventos es descartada. Para probar ambos métodos, vamos a crear una pequeña aplicación que trabaja exactamente como nuestro primer ejemplo, pero también responde a dos comandos específicos: “cerrar1”

y “cerrar2”. Si las cadenas de texto “cerrar1” o “cerrar2” son enviadas desde el formulario, el trabajador será detenido por el código principal o el código del trabajador usando **terminate()** o **close()** respectivamente.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  trabajador=new Worker('trabajador.js');
  trabajador.addEventListener('message', recibido, false);
}
function enviar(){
  var nombre=document.getElementById('nombre').value;
  if(nombre=='cerrar1'){
    trabajador.terminate();
    cajadatos.innerHTML='Trabajador Detenido';
  }else{
    trabajador.postMessage(nombre);
  }
}
function recibido(e){
  cajadatos.innerHTML=e.data;
}
window.addEventListener('load', iniciar, false);
```

Listado 14-7. *Deteniendo el trabajador desde el código principal.*

La única diferencia entre el nuevo código del Listado 14-7 y el del Listado 14-3 es la adición de un condicional **if** para controlar la inserción del comando “cerrar1”. Si este comando es insertado en el formulario en lugar de un nombre, el método **terminate()** es ejecutado y un mensaje es mostrado en pantalla indicando que el trabajador fue detenido. Por el otro lado, si el valor es diferente al comando esperado, el mensaje es enviado al trabajador.

El código para el trabajador realizará una tarea similar. Si el mensaje recibido contiene la cadena de texto “cerrar2”, el trabajador se detendrá a sí mismo usando el método **close()** o enviará una respuesta en caso contrario:

```
addEventListener('message', recibido, false);

function recibido(e){
  if(e.data=='cerrar2'){
    postMessage('Trabajador Detenido');
    close();
  }else{
    var respuesta='Su nombre es '+e.data;
    postMessage(respuesta);
  }
}
```

Listado 14-8. *El trabajador se detiene a sí mismo.*

Hágalo usted mismo: Use el mismo documento HTML y reglas CSS de los Listados 14-1 y 14-2. Copie el código del Listado 14-7 dentro del archivo **webworkers.js** y el código del Listado 14-8 dentro del archivo **trabajador.js**. Abra la plantilla en su navegador y usando el formulario envíe el comando “cerrar1” o “cerrar2”. Luego de ingresar uno de estos comandos el trabajador no responderá más.

APIs síncronas

Los trabajadores pueden presentar limitaciones a la hora de interactuar con el documento principal y acceder a sus elementos, pero cuando se trata de procesamiento y funcionalidad, como ya mencionamos, la situación mejora considerablemente. Por ejemplo, dentro de un trabajador podemos usar los métodos **setTimeout()** o **setInterval()**, cargar información adicional desde el servidor usando **XMLHttpRequest** e incluso utilizar algunas APIs para crear códigos profesionales. Esta última posibilidad es la más prometedora, pero tiene una

trampa: deberemos aprender una implementación diferente para cada una de las APIs a implementar.

Cuando estudiamos algunas APIs, la implementación presentada en esos capítulos era la llamada asíncrona. Muchas APIs ofrecen una versión asíncrona y otra síncrona. Estas diferentes versiones de la misma API realizan las mismas tareas pero usando métodos específicos de acuerdo a la forma en que son procesadas.

Las APIs asíncronas son útiles cuando las operaciones a realizar consumen recursos y tiempo que afectan el normal funcionamiento del documento principal. Las operaciones asíncronas son realizadas detrás de escena mientras el código principal continúa procesándose sin interrupción. Los resultados son informados posteriormente a través de eventos. Debido a que los trabajadores son por naturaleza asíncronos (son procesados al mismo tiempo que el código principal) este tipo de operaciones ya no son necesarias, por lo que el código dentro de un trabajador se ejecuta de forma síncrona (incluyendo las APIs).

> **Hágalo usted mismo:** No vamos a estudiar APIs síncronas en este libro. Varias APIs incluyen una versión síncrona, como API File o API IndexedDB, pero la mayoría de ellas están aún bajo desarrollo o son inestables en este momento. Para mayor información y ejemplos, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

Importando códigos

Algo que vale la pena mencionar es la posibilidad de cargar archivos Javascript externos desde un trabajador. Un trabajador puede contener todo el código necesario para realizar cualquier tarea que se necesite, pero debido a que varios trabajadores pueden ser creados para el mismo documento, existe la posibilidad de que varias partes de este código se repita entre un trabajador y otro y se vuelva así redundante. Podemos seleccionar estos trozos de código y almacenarlos en un simple archivo que será luego cargado por cada trabajador usando el nuevo método **`importScripts()`**:

importScripts(archivo) Este método carga un archivo Javascript externo para incluir código dentro de un trabajador. El atributo **`archivo`** indica la ruta del archivo a ser incluido.

Si alguna vez ha usado métodos en otros lenguajes de programación, seguramente habrá notado la similitud entre **`importScripts()`** y funciones como **`include()`** de PHP, por ejemplo. El código del archivo es incorporado dentro del trabajador y es ejecutado como si fuera parte de su propio código.

Para usar el nuevo método **`importScripts()`** necesitamos declararlo al comienzo del trabajador. El código del trabajador no estará listo para operar hasta que estos archivos sean completamente cargados.

```
importScripts('mascodigos.js');

addEventListener('message', recibido, false);

function recibido(e){
  prueba();
}
```

Listado 14-9. *Cargando códigos Javascript para el trabajador desde archivos externos.*

El código en el Listado 14-9 no es funcional, es solo un ejemplo de cómo aplicar el método **`importScripts()`**. En esta hipotética situación, el archivo **`mascodigos.js`** conteniendo la función **`prueba()`** es cargado tan pronto como el trabajador es iniciado. A partir de este instante, la función **`prueba()`** (así como cualquier otra función dentro del archivo **`mascodigos.js`**) se encuentra disponible para el resto del código del trabajador.

Trabajadores compartidos

Lo que hemos visto hasta el momento es llamado Dedicated Worker (Trabajador Dedicado). Este tipo de trabajador solo responde al código desde el cual fue creado. Existe otro tipo de trabajador llamado Shared Worker (Trabajador Compartido), el cual responde a múltiples documentos desde el mismo origen. Trabajar con múltiples conexiones significa que podemos compartir el mismo trabajador desde diferentes ventanas, pestañas o iframes, y podemos mantener a cada instancia actualizada y sincronizada para la construcción de aplicaciones complejas.

Las conexiones son hechas a través de puertos y estos puertos pueden ser almacenados en el trabajador para futuras referencias. Para trabajar con Trabajadores Compartidos y puertos, esta parte de la API incorpora

nuevas propiedades, métodos y eventos:

SharedWorker(códigoURL) Este constructor reemplaza al constructor previo **`Worker()`** usado para Trabajadores Dedicados. Como siempre, el atributo **`códigoURL`** declara la ruta del archivo Javascript con el código para el trabajador. Un segundo atributo opcional puede ser agregado para especificar el nombre del trabajador.

port Cuando el objeto SharedWorker es construido, un nuevo puerto es creado para este documento y asignado a la propiedad **`port`**. Esta propiedad será usada más adelante para referenciar el puerto y comunicarse con el trabajador.

connect Este es un evento específico usado desde dentro del trabajador para controlar nuevas conexiones. El evento será disparado cada vez que un documento inicia una conexión con el trabajador. Es útil para seguir los pasos de todas las conexiones disponibles con el trabajador (para referenciar todos los documentos que lo están usando).

start() Este método está disponible para los objetos MessagePort (uno de los objetos retornados durante la construcción del trabajador compartido) y su función es iniciar el envío de mensajes recibidos en un puerto. Luego de la construcción del objeto SharedWorker, este método debe ser llamado para iniciar la conexión.

El constructor **`SharedWorker()`** retorna un objeto SharedWorker y un objeto MessagePort con el valor del puerto a través del cual la conexión con el trabajador será hecha. La comunicación con el trabajador debe ser realizada por medio del puerto referenciado por la propiedad **`port`**.

Para estudiar el funcionamiento de los Trabajadores Compartidos, tendremos que usar al menos dos documentos diferentes ubicados en el mismo origen, un archivo Javascript para cada documento y un archivo más para el trabajador.

La plantilla para nuestro ejemplo incluye un iframe donde se cargará el segundo documento HTML. Ambos, el documento principal y el documento dentro del iframe, compartirán el mismo trabajador.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>WebWorkers</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
<body>
  <section id="cajaformulario">
    <form name="formulario">
      <p>Nombre:<br><input type="text" name="nombre"
                                                 id="nombre"></p>
      <p><input type="button" name="boton" id="boton"
                                              value="Enviar"></p>
    </form>
  </section>
  <section id="cajadatos">
    <iframe id="iframe" src="iframe.html" width="500"
                                           height="350"></iframe>
  </section>
</body>
</html>
```

Listado 14-10. *Plantilla para usar Trabajadores Compartidos.*

El documento para el iframe es un simple documento HTML que incluye un elemento **`<section>`** para definir nuestra ya conocida **`cajadatos`** y el archivo **`iframe.js`** con el código necesario para realizar la conexión con el trabajador:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>iframe</title>
  <script src="iframe.js"></script>
```

```
</head>
<body>
  <section id="cajadatos"></section>
</body>
</html>
```

Listado 14-11. *Plantilla para el iframe (`iframe.html`).*

Cada uno de los documentos HTML creados incluye su propio código Javascript para iniciar la conexión con el trabajador y procesar sus respuestas. Estos códigos deben construir el objeto SharedWorker y usar el puerto referenciado por el valor de la propiedad **`port`** para enviar y recibir mensajes. Veamos primero el código para el documento principal:

```
function iniciar(){
  var boton=document.getElementById('boton');
  boton.addEventListener('click', enviar, false);

  trabajador=new SharedWorker('trabajador.js');
  trabajador.port.addEventListener('message', recibido, false);
  trabajador.port.start();
}
function recibido(e){
  alert(e.data);
}
function enviar(){
  var nombre=document.getElementById('nombre').value;
  trabajador.port.postMessage(nombre);
}
window.addEventListener('load', iniciar, false);
```

Listado 14-12. *Conectando desde el documento principal (`webworkers.js`).*

Cada uno de los documentos que necesita utilizar el trabajador compartido deberá crear un objeto SharedWorker y establecer la conexión con el trabajador. En el código del Listado 14-12, el objeto es construido usando el archivo **`trabajador.js`** como el archivo que contendrá al trabajador, y luego las operaciones de comunicación son hechas a través del puerto correspondiente usando la propiedad **`port`**.

Luego de que una escucha para el evento **`message`** es agregada para escuchar por respuestas provenientes del trabajador, el método **`start()`** es llamado para iniciar el proceso de envío de mensajes. La conexión con el trabajador compartido no es establecida hasta que este método es ejecutado (a menos que usemos manejadores de eventos, como **`onmessage`**, en lugar del método **`addEventListener()`**).

La función **`enviar()`** es similar a ejemplos previos, excepto que esta vez la comunicación es realizada a través del valor de la propiedad **`port`**.

El código del iframe se asemeja al principal:

```
function iniciar(){
  trabajador=new SharedWorker('trabajador.js');
  trabajador.port.addEventListener('message', recibido, false);
  trabajador.port.start();
}
function recibido(e){
  var cajadatos=document.getElementById('cajadatos');
  cajadatos.innerHTML=e.data;
}
window.addEventListener('load', iniciar, false);
```

Listado 14-13. *Conectando desde el iframe (`iframe.js`).*

En ambos códigos, el objeto SharedWorker es construido referenciando al mismo archivo (**`trabajador.js`**), y la conexión es establecida usando la propiedad **`port`** (aunque a través de puertos diferentes). La única diferencia entre el código para el documento principal y el código para el iframe es cómo la respuesta del trabajador será procesada. En el documento principal, la función **`recibido()`** muestra una ventana de alerta (ver Listado 14-12),

mientras que dentro del iframe la respuesta es mostrada como un simple texto dentro del elemento **cajadatos** (ver Listado 14-13).

Es momento de ver cómo el trabajador compartido administra cada conexión y envía mensajes en respuesta al documento apropiado. Recuerde que solo existe un trabajador para ambos documentos (precisamente por esto se llama Trabajador Compartido). Debido a esto, cada solicitud de conexión recibida por el trabajador tiene que ser diferenciada y almacenada para futuras referencias. Vamos a grabar estas referencias a los puertos de cada documento en un array llamado **puertos**:

```
puertos=new Array();

addEventListener('connect', conectar, false);

function conectar(e){
  puertos.push(e.ports[0]);
  e.ports[0].onmessage=enviar;
}
function enviar(e){
  for(f=0; f < puertos.length; f++){
    puertos[f].postMessage('Su nombre es '+e.data);
  }
}
```

Listado 14-14. *Código para el trabajador compartido (`trabajador.js`).*

El procedimiento es similar al usado para Trabajadores Dedicados. Esta vez solo tenemos que considerar a qué documento específico vamos a responder, porque varios pueden estar conectados con el trabajador al mismo tiempo. Con este propósito, el evento **connect** provee el array **ports** con el valor del nuevo puerto creado para la conexión que estamos estableciendo (el array contiene solo este valor localizado en el índice 0).

Cada vez que un código solicita una nueva conexión al trabajador, el evento **connect** es disparado. En el código del Listado 14-14, este evento llama a la función **conectar()**. En esta función realizamos dos operaciones: primero, el valor del puerto es tomado de la propiedad **ports** (índice 0) y almacenado en el array **puertos** (inicializado al comienzo del código del trabajador). Y segundo, el manejador de eventos **onmessage** es registrado para este puerto en particular y la función **enviar()** es declarada como la responsable de atender los mensajes recibidos.

Como resultado, cada vez que un mensaje es enviado hacia el trabajador desde el código principal, sin importar desde qué documento, la función **enviar()** en el trabajador es ejecutada. En esta función usamos un bucle **for** para extraer del array **puertos** todos los puertos abiertos para este trabajador y enviar un mensaje a cada documento conectado. El proceso es exactamente como en los Trabajadores Dedicados, pero esta vez varios documentos son respondidos en lugar de uno solo.

> **Hágalo usted mismo:** Para probar este ejemplo, tendrá que crear varios archivos y subirlos al servidor. Cree un archivo HTML con la plantilla del Listado 14-10 y el nombre que desee. Esta plantilla usará el mismo archivo **webworkers.css** usado a lo largo del capítulo. Cree también el archivo **webworkers.js** con el código del Listado 14-12, y el archivo **iframe.html** como la fuente del iframe con el código del Listado 14-11. También necesitará crear el archivo **trabajador.js** conteniendo el código del trabajador del Listado 14-14. Una vez que todos estos archivos son generados y subidos al servidor, abra el primer documento en su navegador. Use el formulario para enviar un mensaje al trabajador y ver cómo ambos documentos (el documento principal y el documento dentro del iframe) procesan la respuesta.

> **IMPORTANTE:** En el momento de escribir estas líneas, los Trabajadores Compartidos solo funcionan en navegadores basados en el motor WebKit, como Google Chrome y Safari.

14.2 Referencia rápida

La API Web Workers incorpora la capacidad de multiprocesamiento a Javascript. Es la API que nos permite procesar códigos detrás de escena sin interrumpir el normal funcionamiento del código del documento principal.

Trabajadores

Dos clases diferentes de trabajadores son ofrecidos: Trabajadores Dedicados (Dedicated Workers) y Trabajadores Compartidos (Shared Workers). Ambos comparten los siguientes métodos y eventos:

postMessage(mensaje) Este método envía un mensaje al trabajador, el código principal o el puerto correspondiente. El atributo `mensaje` es la cadena de texto o el objeto JSON a ser enviado.

terminate() Este método detiene el trabajador desde el código principal.

close() Este método detiene al trabajador desde dentro del trabajador.

importScripts(archivo) Este método carga un archivo Javascript externo para incorporar código al trabajador. El atributo `archivo` indica la ruta del archivo a ser incluido.

message Este evento es disparado cuando un mensaje es enviado al código. Puede ser usado en el trabajador para escuchar por mensajes provenientes del código principal o viceversa.

error Este evento es disparado cuando ocurre un error en el trabajador. Es usado en el código principal para controlar por errores en el trabajador. Retorna información por medio de tres propiedades: `message`, `filename` y `lineno`. La propiedad `message` representa el mensaje de error, la propiedad `filename` muestra el nombre del archivo con el código que causó el error, y la propiedad `lineno` retorna el número de línea en la cual ocurrió el error.

Trabajadores dedicados (Dedicated Workers)

Los Trabajadores Dedicados tienen su propio constructor:

Worker(códigoURL) Este constructor retorna un objeto Worker. El atributo `códigoURL` es la ruta del archivo conteniendo el trabajador.

Trabajadores compartidos (Shared Workers)

Debido a la naturaleza de los Trabajadores Compartidos, la API ofrece algunos métodos, propiedades y eventos específicos:

SharedWorker(códigoURL) Este constructor retorna un objeto SharedWorker. El atributo `códigoURL` es la ruta del archivo conteniendo el trabajador compartido. Un segundo atributo opcional puede ser especificado para declarar el nombre del trabajador.

port Esta propiedad retorna el valor del puerto de la conexión con el trabajador compartido.

connect Este evento es disparado en el trabajador compartido cuando una nueva conexión es solicitada desde un documento.

start() Este método inicia el envío de mensajes. Es usado para comenzar la conexión con el trabajador compartido.

Capítulo 15
API History

15.1 Interface History

Lo que en HTML5 normalmente llamamos API History es en realidad solo una mejora de una vieja API que nunca tuvo una implementación oficial pero fue soportada por navegadores durante años. Esta vieja API estaba compuesta solo por un pequeño grupo de métodos y propiedades, algunos de ellos parte del objeto History. La nueva API History es precisamente una mejora de este objeto y fue incluida oficialmente en la especificación HTML como la interface History. Esta interface combina todos los viejos métodos y propiedades con algunos nuevos para trabajar y modificar el historial del navegador de acuerdo a nuestras necesidades.

Navegando por la Web

El historial del navegador es una lista de todas las páginas web (URLs) visitadas por el usuario durante una sesión. Es lo que hace la navegación posible. Usando los botones de navegación a la izquierda de la barra de navegación de todo navegador podemos ir hacia atrás o hacia adelante en este lista y visitar documentos que vimos anteriormente. Esta lista es construida con URLs reales generadas por los sitios web, incluidas en cada enlace dentro de sus documentos. Con las flechas del navegador podemos cargar la página web que fue visitada anteriormente o volver a la última.

A pesar de la practicidad de los botones de navegación, a veces es útil navegar a través del historial desde dentro del documento. Para simular las flechas de navegación desde Javascript, siempre contamos con los siguientes métodos y propiedades:

back() Este método retrocede un paso en el historial (imitando la flecha izquierda del navegador).

forward() Este método avanza un paso en el historial (imitando la flecha derecha del navegador).

go(pasos) Este método avanza o retrocede en el historial la cantidad de pasos especificados. El atributo `pasos` puede ser un valor negativo o positivo de acuerdo a la dirección hacia dónde queremos ir.

length Esta propiedad retorna el número de entradas en el historial (el total de URLs en la lista).

Estos métodos y propiedades deben ser declarados como parte del objeto History, con una expresión como `history.back()`. También podemos usar el objeto Window para referenciar la ventana, pero esto no es necesario. Por ejemplo, si queremos regresar a la página anterior en el historial podemos usar los códigos `window.history.back()` o `window.history.go(-1)`.

IMPORTANTE: Esta parte de la API es conocida y utilizada por la mayoría de los diseñadores y programadores web estos días. No vamos a estudiar ningún código de ejemplo sobre estos métodos, pero puede visitar nuestro sitio web y seguir los enlaces correspondientes a este capítulo si necesita mayor información al respecto.

Nuevos métodos

Cuando el uso del objeto XMLHttpRequest se volvió estándar y las aplicaciones Ajax se convirtieron en un éxito extraordinario, la forma en la que los usuarios navegaban y accedían a los documentos cambió para siempre. Ahora es común programar pequeños códigos para obtener información desde el servidor y mostrarla dentro del documento actual, sin actualizar el contenido completo de la ventana o cargar un nuevo documento. Los usuarios interactúan con sitios web modernos y aplicaciones desde la misma URL, recibiendo información, ingresando datos y obteniendo resultados de procesos siempre desde la misma página web. La web ha comenzado a emular aplicaciones de escritorio.

Sin embargo, la forma en la que los navegadores siguen los pasos del usuario es a través de URLs. URLs son, de hecho, los datos dentro de la lista de navegación, las direcciones que indican dónde el usuario se encuentra actualmente. Debido a que las nuevas aplicaciones web evitan el uso de URLs para apuntar a la ubicación del usuario dentro de la aplicación, pronto se volvió evidente que pasos importantes en el proceso se perdían sin dejar rastro alguno. Los usuarios podían actualizar datos en una página web docenas de veces y aun así ningún rastro de actividad quedaba almacenado en el historial del navegador para indicar los pasos seguidos.

Nuevos métodos y propiedades fueron incorporados a la ya existente History API con la intención de modificar

manualmente la URL en la barra de localización así como también el historial del navegador simplemente usando código Javascript. Desde ahora tenemos la posibilidad de agregar URLs falsas al historial y de este modo mantener control sobre la actividad del usuario.

pushState(estado, título, url) Este método crea una nueva entrada en el historial. El atributo **estado** declara un valor para el estado de la entrada. Es útil para identificar la entrada más adelante y puede ser especificado como una cadena de texto o un objeto JSON. El atributo **título** es el título de la entrada, y el atributo **url** es la URL para la entrada que estamos generando en el historial (este valor reemplazará a la URL que aparece actualmente en la barra de localización).

replaceState(estado, título, url) Este método trabaja exactamente igual a **pushState()**, pero no genera una nueva entrada. En su lugar, reemplaza la información de la actual.

state Esta propiedad retorna el valor del estado de la entrada actual. Este valor será **null** (nulo) a menos que haya sido declarado por alguno de los métodos anteriores usando el atributo **estado**.

URLs falsas

Las URLs generadas usando métodos como **pushState()** son URLs falsas en el sentido de que los navegadores nunca controlan su validez y la existencia del documento al que supuestamente apuntan. Depende de nosotros asegurarnos que estas URLs falsas sean en realidad válidas y útiles.

Para crear una nueva entrada en el historial del navegador y cambiar la dirección URL dentro de la barra de navegación, necesitamos usar el método **pushState()**. Veamos un ejemplo de cómo trabaja:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>History API</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="contenido">
    Este contenido nunca es actualizado<br>
    <span id="url">página 2</span>
  </section>
  <aside id="cajadatos"></aside>
</body>
</html>
```

Listado 15-1. *Plantilla básica para aplicar la API History.*

En el Listado 15-1 presentamos un código HTML con los elementos básicos necesarios para probar esta API. Con este propósito, colocamos contenido permanente dentro de un elemento **<section>** identificado como **contenido**, un texto que se convertirá en un link para generar la segunda página virtual, y nuestra ya acostumbrada **cajadatos** para mostrar el contenido alternativo.

Los siguientes son los estilos básicos necesarios para diferenciar las partes que componen la plantilla:

```
#contenido{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#cajadatos{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#contenido span{
```

```
    color: #0000FF;
    cursor: pointer;
  }
```

Listado 15-2. *Estilos para las cajas y los elementos* `<span>` *(*`history.css`*).*

Lo que vamos a hacer en este ejemplo es agregar una nueva entrada con el método **`pushState()`** y actualizar el contenido sin recargar la página o cargar un nuevo documento.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  url=document.getElementById('url');
  url.addEventListener('click', cambiar, false);
}
function cambiar(){
  cajadatos.innerHTML='La URL es pagina2';
  window.history.pushState(null, null, 'pagina2.html');
}
window.addEventListener('load', iniciar, false);
```

Listado 15-3. *Generando una nueva URL y nuevo contenido (*`history.js`*).*

En la función **`iniciar()`** del Listado 15-3, creamos la referencia apropiada para el elemento **`cajadatos`** y agregamos una escucha para el evento **`click`** al elemento **`<span>`**. Cada vez que el usuario hace clic sobre el texto dentro de **`<span>`**, la función **`cambiar()`** es llamada.

La función **`cambiar()`** realiza dos tareas: actualiza el contenido de la página con nueva información e inserta una nueva URL al historial. Luego de que esta función es ejecutada, **`cajadatos`** muestra el texto ”La URL es pagina2” y la URL del documento principal en la barra de localización es reemplazada por la URL falsa ”pagina2.html”.

Los atributos **`estado`** y **`título`** para el método **`pushState()`** esta vez fueron declarados como **`null`** (nulo). El atributo **`título`** no está siendo usado en este momento por ningún navegador y siempre lo declararemos como **`null`**, pero el atributo **`estado`**, por el contrario, es útil y será aprovechado en próximos ejemplos.

> **Hágalo usted mismo:** Copie la plantilla en el Listado 15-1 dentro de un archivo HTML. Cree un archivo CSS llamado **`history.css`** con los estilos del Listado 15-2 y un archivo Javascript llamado **`history.js`** con los códigos del Listado 15-3. Súbalos a su servidor y abra el archivo HTML en su navegador. Haga clic sobre el texto “página 2” y compruebe que la URL en la barra de localización cambió por la URL falsa generada por el código.

Siguiendo la pista

Lo que hemos hecho hasta ahora es solo una manipulación del historial de la sesión. Le hicimos creer al navegador que el usuario visitó una URL que, a este punto, ni siquiera existe. Luego de que el usuario hace clic en el enlace “página 2”, la URL falsa “pagina2.html” es mostrada en la barra de localización y nuevo contenido es insertado en el elemento **`cajadatos`**, todo sin recargar la página web o cargar una nueva. Es un truco interesante pero no realmente útil. El navegador todavía no considera a la nueva URL como un documento real. Si intenta retroceder o avanzar en el historial usando los botones de navegación del navegador, la URL cambia entre la que generamos artificialmente y la URL del documento principal, pero el contenido del documento no es modificado. Necesitamos detectar cuando las URLs falsas son visitadas nuevamente y realizar las modificaciones apropiadas al documento para mostrar el estado correspondiente a la URL actual.

Previamente mencionamos la existencia de la propiedad **`state`**. El valor de esta propiedad es declarado durante la generación de la nueva URL y es usado para identificar cuál es la dirección web actual. Para trabajar con esta propiedad, la API provee un nuevo evento:

popstate Este evento es disparado cuando una URL es visitada nuevamente o un documento es cargado. Provee la propiedad **`state`** con el valor del estado declarado cuando la URL fue generada con los métodos **`pushState()`** o **`replaceState()`**. Este valor es **`null`** (nulo) si la URL es real, a menos que lo hayamos cambiado antes usando **`replaceState()`**, como veremos en el siguiente ejemplo.

En el próximo código mejoraremos el ejemplo previo implementando el evento **`popstate`** y el método **`replaceState()`** para detectar cuál URL el usuario está solicitando a cada momento.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  url=document.getElementById('url');
  url.addEventListener('click', cambiar, false);
  window.addEventListener('popstate', nuevaurl ,false);
  window.history.replaceState(1, null);
}
function cambiar(){
  mostrar(2);
  window.history.pushState(2, null, 'pagina2.html');
}
function nuevaurl(e){
  mostrar(e.state);
}
function mostrar(actual){
  cajadatos.innerHTML='La URL es página '+actual;
}
window.addEventListener('load', iniciar, false);
```

Listado 15-4. *Controlando la ubicación del usuario (`history.js`).*

Debemos hacer dos cosas en nuestra aplicación para tener control absoluto sobre la situación. Primero, tenemos que declarar un valor de estado para cada URL que vamos a utilizar, las falsas y las reales. Y segundo, el contenido del documento debe ser actualizado de acuerdo a la URL actual.

En la función **`iniciar()`** del Listado 15-4, una escucha fue agregada para el evento **`popstate`**. Esta escucha llamará a la función **`nuevaurl()`** cada vez que una URL es visitada nuevamente. La función simplemente actualizará el contenido de la **`cajadatos`** con un mensaje indicando cuál es la página actual. Lo que hace es tomar el valor de la propiedad **`state`** y enviarlo a la función **`mostrar()`** para mostrarlo en pantalla.

Esto funcionará para cada URL falsa, pero como explicamos antes, las URLs reales no tienen un valor de estado por defecto. Usando el método **`replaceState()`** al final de la función **`iniciar()`** cambiamos la información de la entrada actual (la URL real del documento principal) y declaramos el valor 1 para su estado. Ahora, cada vez que el usuario visite nuevamente el documento principal podremos detectarlo comprobando este valor.

La función **`cambiar()`** es la misma que antes, excepto que esta vez usa la función **`mostrar()`** para actualizar el contenido del documento y declarar el valor 2 para el estado de la URL falsa.

La aplicación trabaja de la siguiente forma: cuando el usuario hace clic sobre el enlace "página 2", el mensaje "La URL es página 2" es mostrado en pantalla y la URL en la barra de navegación es reemplazada por "pagina2.html" (incluyendo la ruta completa, por supuesto). Esto es lo que habíamos hecho hasta el momento, pero aquí es donde las cosas se vuelven interesantes. Si el usuario presiona la flecha izquierda en la barra de navegación del navegador, la URL dentro de la barra de localización será reemplazada por la que se encuentra en la posición anterior del historial (esta es la URL real de nuestro documento) y el evento **`popstate`** será disparado. Este evento llama a la función **`nuevaurl()`** que lee el valor de la propiedad **`state`** y lo envía a la función **`mostrar()`**. Ahora el valor del estado es 1 (el valor que declaramos para esta URL usando el método **`replaceState()`**) y el mensaje mostrado en la pantalla será "La URL es página 1". Si el usuario vuelve a visitar la URL falsa usando la flecha derecha en la barra de navegación, el valor del estado será 2 y el mensaje mostrado en pantalla será nuevamente "La URL es página 2".

Como puede ver, el valor de la propiedad **`state`** es cualquier valor que desee usar para controlar cuál es la URL actual y adaptar el contenido del documento a la misma.

Hágalo usted mismo: Use los archivos con los códigos de los Listados 15-1 y 15-2 para el documento HTML y los estilos CSS. Copie el código del Listado 15-4 dentro del archivo **`history.js`** y suba todos los archivos a su servidor. Abra la plantilla HTML en su navegador y haga clic sobre el texto "página 2". La nueva URL será mostrada y el contenido de **`cajadatos`** cambiará de acuerdo a la URL correspondiente. Presione las flechas izquierda y derecha en el navegador para moverse a través del historial y ver cómo la URL cambia y cómo el contenido del documento es actualizado de acuerdo a la URL seleccionada (el contenido es mostrado en pantalla de acuerdo al valor del estado actual).

IMPORTANTE: La URL "pagina2.html" generada con el método **`pushState()`** en los ejemplos previos es considerada falsa, pero debería ser real. El propósito de esta API no es crear URLs falsas sino proveer a los programadores la alternativa de registrar la actividad del usuario en el historial para poder volver a un estado anterior toda vez que sea requerido (incluso luego de que el navegador fue cerrado). Usted mismo deberá

asegurarse de que el código en su servidor retorna el apropiado contenido por cada una de las URLs usadas por la aplicación (las reales y las falsas).

Ejemplo real

La siguiente es una aplicación práctica. Vamos a usar la API History y todos los métodos estudiados anteriormente para cargar un grupo de cuatro imágenes desde el mismo documento. Cada imagen es asociada a una URL falsa que podrá ser usada más adelante para retornar una imagen específica desde el servidor.

El documento principal es cargado con una imagen por defecto. Esta imagen estará asociada al primero de cuatro enlaces que son parte del contenido permanente del documento. Todos estos enlaces apuntarán a URLs falsas referenciando un estado, no un documento real (incluyendo el enlace para el documento principal que será cambiado por "pagina1.html"). Todo el proceso tendrá más sentido pronto, por ahora veamos el código de la plantilla HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>History API</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="contenido">
    Este contenido nunca es actualizado <br>
    <span id="url1">imagen 1</span> -
    <span id="url2">imagen 2</span> -
    <span id="url3">imagen 3</span> -
    <span id="url4">imagen 4</span> -
  </section>
  <aside id="cajadatos">
    <img id="imagen" src="http://www.minkbooks.com/content/
                                              monster1.gif">
  </aside>
</body>
</html>
```

Listado 15-5. *Plantilla para una aplicación "real".*

La única diferencia significativa entre esta nueva aplicación y la anterior es el número de enlaces y la cantidad de URLs que estamos manejando. En el código del Listado 15-4, teníamos dos estados, el estado 1 correspondiente al documento principal y el estado 2 para la URL falsa "pagina2.html" generada por el método **`pushState()`**. En este caso, debemos automatizar el proceso y generar un total de cuatro URLs falsas correspondientes a cada imagen disponible.

```
function iniciar(){
  for(var f=1;f<5;f++){
    url=document.getElementById('url'+f);
    url.addEventListener('click', function(x){
        return function(){ cambiar(x);}
      }(f), false);
  }

  window.addEventListener('popstate', nuevaurl ,false);
  window.history.replaceState(1, null, 'pagina1.html');
}
function cambiar(pagina){
  mostrar(pagina);
  window.history.pushState(pagina, null, 'pagina'+pagina+'.html');
}
function nuevaurl(e){
  mostrar(e.state);
```

```
}
function mostrar(actual){
  if(actual!=null){
    imagen=document.getElementById('imagen');
    imagen.src='http://www.minkbooks.com/content/monster' +
                                                  actual + '.gif';
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 15-6. *Manipulando el historial (`history.js`).*

Como se puede apreciar, estamos usando las mismas funciones pero con algunos cambios importantes. Primero, el método **`replaceState()`** en la función **`iniciar()`** tiene el atributo **`url`** declarado como "pagina1.html". Decidimos programar nuestra aplicación de este modo, declarando el estado del documento principal como 1 y cambiando su URL por “pagina1.html” (independientemente de la URL real del documento). De este modo será simple pasar de un estado a otro, siempre usando el mismo formato y los valores de la propiedad **`state`** para construir todas las URL utilizadas por la aplicación. Puede ver este procedimiento en la práctica estudiando la función **`cambiar()`**. Cada vez que el usuario hace clic en uno de los enlaces de la plantilla, esta función es ejecutada y la URL falsa es construida con el valor de la variable **`pagina`** y agregada al historial de la sesión. El valor recibido por esta función fue previamente declarado en el bucle **`for`** al comienzo de la función **`iniciar()`**. Este valor es declarado como 1 para el enlace “página 1”, 2 para el enlace “página 2”, y así sucesivamente.

Cada vez que una URL es visitada, la función **`mostrar()`** es ejecutada para actualizar el contenido (la imagen) de acuerdo a la misma. Debido a que el evento **`popstate`** a veces es disparado en circunstancias en las que el valor de la propiedad **`state`** es **`null`** (como cuando el documento es cargado por primera vez), controlamos el valor recibido por la función **`mostrar()`** antes de continuar. Si este valor es diferente de **`null`**, significa que la propiedad **`state`** fue definida para esa URL, por lo tanto la imagen correspondiente con ese estado es mostrada en pantalla.

Las imágenes usadas para este ejemplo fueron nombradas monster1.gif, monster2.gif, monster3.gif y monster4.gif, siguiendo el mismo orden de los valores de la propiedad **`state`**. Así, usando este valor podemos seleccionar la imagen a ser mostrada. Sin embargo, recuerde que los valores usados pueden ser cualquiera que usted necesite y el proceso para crear URLs falsas y contenido asociado debe ser desarrollado de acuerdo con las necesidades de su aplicación.

También recuerde que los usuarios deberían poder regresar a cualquiera de las URLs generadas por la aplicación y ver el contenido correspondiente en la pantalla cada vez que lo deseen. Usted debe preparar su servidor para procesar estas URLs de modo que cada estado de la aplicación (cada URL falsa) esté disponible y sea siempre accesible. Por ejemplo, si un usuario abre una nueva ventana y escribe la URL “pagina2.html” en la barra de navegación, el servidor debería retornar el documento principal conteniendo la imagen “monster2.gif”, correspondiente a esta URL, y no simplemente la plantilla del Listado 15-5. La idea detrás de esta API es proveer una alternativa para que los usuarios puedan regresar a cualquier estado previo, en cualquier momento; algo que solo podemos lograr volviendo válidas a las URLs falsas.

> **IMPORTANTE:** El bucle **`for`** usado en el código del Listado 15-6 para agregar una escucha para el evento **`click`** a cada elemento **`<span>`** en el documento aprovecha una técnica Javascript que nos permite enviar valores reales a una función. Para enviar un valor a la función que manejará el evento en un método **`addEventListener()`**, debemos declarar el valor real. Si en su lugar enviamos una variable, lo que realmente es enviado no es el valor de la variable sino una referencia a la misma. Por lo tanto, en este caso, para enviar el valor actual de la variable **`f`** en el bucle **`for`** tenemos que usar varias funciones anónimas. La primera función es ejecutada en el momento en el que el método **`addEventListener()`** es declarado. Esta función recibe el valor actual de la variable **`f`** (vea los paréntesis al final) y almacena este valor en la variable **`x`**. Luego, la función retorna una segunda función con el valor de la variable **`x`**. Esta segunda función es la que será ejecutada con el valor correspondiente cuando el evento es disparado. Esta es una técnica compleja que deberá aprender junto con otras si desea crear códigos Javascript profesionales. Para obtener más información sobre este tema, visite nuestro sitio web y siga los enlaces correspondientes a este capítulo.

> **Hágalo usted mismo:** Para probar el último ejemplo use el mismo documento HTML del Listado 15-5 con los estilos CSS del Listado 15-2. Copie el código del Listado 15-6 dentro del archivo **`history.js`** y suba los archivos a su servidor. Abra la plantilla en su navegador y haga clic en los enlaces. Navegue a través de las URLs seleccionadas usando los botones de navegación del navegador. Las imágenes en la pantalla cambiarán de acuerdo a la URL en la barra de localización.

15.2 Referencia rápida

API History nos permite manipular el historial de la sesión en el navegador para seguir los pasos de los usuarios dentro de la aplicación. Esta API es incluida en la especificación oficial como la interface History. Esta interface combina métodos y propiedades, nuevos y viejos.

length Esta propiedad retorna el número total de entradas en el historial.

state Esta propiedad retorna el valor del estado para la URL actual.

go(pasos) Este método avanza o retrocede en el historial de navegación de acuerdo al valor del atributo `paso`. Este valor puede ser negativo o positivo dependiendo de la dirección de navegación deseada.

back() Este método carga la URL anterior desde el historial.

forward() Este método carga la URL siguiente desde el historial.

pushState(estado, título, url) Este método inserta nuevos datos en el historial. El atributo `estado` es el valor del estado que queremos otorgar a esta nueva entrada. El atributo `título` es el título de la entrada. Y el atributo `url` es la nueva URL que queremos generar en el historial.

replaceState(estado, título, url) Este método modifica la entrada actual. El atributo `estado` es el valor del estado que queremos otorgar a la entrada actual. El atributo `título` es el título de la entrada. Y el atributo `url` es la nueva URL que queremos asignar a la entrada actual.

popstate Este evento es disparado en determinadas circunstancias para informar el valor del estado actual.

Capítulo 16
API Offline

16.1 Caché

Los días de trabajar desconectado han llegado a su fin. Debido a que este capítulo presenta API Offline (la API para trabajar desconectados), esta declaración puede resultar contradictoria, pero analicémoslo por un momento. Hemos trabajado desconectados casi toda nuestra vida. Las aplicaciones de escritorio fueron nuestra herramienta primaria de producción. Y ahora, de repente, la web ha emergido como la nueva plataforma de trabajo. Aplicaciones en línea se vuelven más y más complejas, y HTML5 está haciendo la batalla entre estos dos mundos más dura que nunca. Bases de datos, acceso a archivos, almacenamiento, herramientas gráficas, edición de imagen y video, y multiprocesamiento son, entre otras, características esenciales para una aplicación que ahora se encuentran disponibles en la web. Nuestra actividad diaria gira cada vez más en torno a la web, y nuestro ámbito de producción se encuentra en la red. Los días de trabajar desconectados son historia.

Sin embargo, a medida que esta transición continúa, las aplicaciones web se vuelven más sofisticadas, requiriendo archivos más grandes y mayor tiempo de descarga. Para cuando las aplicaciones en la web reemplacen definitivamente a las aplicaciones de escritorio, trabajar en línea será imposible. Los usuarios no podrán descargar varios megabytes de archivos cada vez que necesiten usar una aplicación y no podrán contar con tener conexión a la red disponible todo el tiempo. Las aplicaciones que no requieren Internet pronto desaparecerán, pero bajo las actuales circunstancias las aplicaciones en línea están destinadas a fracasar.

Offline API llega para ayudarnos a resolver este dilema. Básicamente, esta API provee la alternativa de almacenar las aplicaciones y archivos web en el ordenador del usuario para uso futuro. Un solo acceso es suficiente para descargar todos los archivos requeridos y ejecutar la aplicación en todo momento, con o sin conexión a Internet. Una vez que los archivos son descargados, la aplicación funciona en el navegador usando este caché (los archivos almacenados en el ordenador), como lo haría una aplicación de escritorio, independientemente de lo que pase con el servidor o la conexión.

El archivo manifiesto

Una aplicación web o un sitio web sofisticado consistirá en varios archivos, pero no todos ellos serán requeridos para ejecutar la aplicación y tampoco será necesario almacenarlos a todos en el ordenador del usuario. La API asigna un archivo específico para declarar la lista de archivos que la aplicación necesita para trabajar sin conexión. Este es solo un archivo de texto llamado “manifiesto” (*manifest*), conteniendo una lista de URLs que apuntan a la ubicación de los archivos requeridos. El manifiesto puede ser creado con cualquier editor de texto, y debe comenzar con la línea **`CACHE MANIFEST`**, como en el siguiente ejemplo:

```
CACHE MANIFEST
cache.html
cache.css
cache.js
```

Listado 16-1. *Archivo manifiesto.*

El manifiesto deberá ser grabado con la extensión **`.manifest`** y deberá incluir debajo de **`CACHE MANIFEST`** todos los archivos que la aplicación necesita para trabajar desde el ordenador del usuario sin solicitar ningún recurso externo. En nuestro ejemplo, tenemos el archivo **`cache.html`** como el documento principal de la aplicación, el archivo **`cache.css`** con los estilos CSS y el archivo **`cache.js`** conteniendo los códigos Javascript.

Categorías

Del mismo modo que necesitamos especificar los archivos requeridos para trabajar desconectados, también podríamos necesitar declarar explícitamente aquellos que se encuentran solo disponibles cuando estamos conectados. Este podría ser el caso para algunas partes de la aplicación que solo serán útiles cuando tenemos acceso a Internet (por ejemplo, una sala de chat para consultas).

Para identificar los tipos de archivos listados en el archivo manifiesto, la API introduce tres categorías:

CACHE Esta es la categoría por defecto. Todos los archivos en esta categoría serán almacenados en el ordenador del usuario para uso futuro.

NETWORK Esta categoría es considerada como una lista de aprobación; todos los archivos en su interior solo se encuentran disponibles en línea.

FALLBACK Esta categoría es para archivos que podría ser útil obtener del servidor cuando estamos conectados, pero que pueden ser reemplazados por una versión en el caché. Si el navegador detecta que hay conexión, intentará usar el archivo original en el servidor, en caso contrario, será usado en su lugar el alternativo ubicado en el ordenador del usuario.

Usando categorías, nuestro archivo manifiesto podría ser similar al siguiente:

```
CACHE MANIFEST

CACHE:
cache.html
cache.css
cache.js

NETWORK:
chat.html

FALLBACK:
noticias.html sinnoticias.html
```

Listado 16-2. *Declarando archivos por categoría.*

En el nuevo archivo manifiesto del Listado 16-2, los archivos son listados bajo la categoría correspondiente. Los tres archivos en la categoría **CACHE** serán descargados, almacenados en el ordenador del usuario y usados para esta aplicación de ahora en más (a menos que especifiquemos algo diferente más adelante). El archivo **chat.html** especificado en la categoría **NETWORK** estará solo disponible cuando el navegador tenga acceso a Internet. Y por último, el archivo **noticias.html** dentro de la categoría **FALLBACK** será accedido desde el servidor cuando exista conexión a la red, o reemplazado por el archivo **sinnoticias.html** ubicado en el ordenador del usuario en caso contrario. Del mismo modo que los archivos dentro de la categoría **CACHE**, el archivo **sinnoticias.html** es incluido en el caché y por lo tanto almacenado en el ordenador del usuario para estar disponible cuando sea requerido.

La categoría **FALLBACK** es útil no solo para reemplazar archivos individuales sino también para proveer alternativas para directorios completos. Por ejemplo, la línea **/ sinconexion.html** reemplazará cualquier archivo que no esté disponible en el caché por el archivo **sinconexion.html**. Esta es una forma simple de desviar a los usuarios hacia un documento que les recomienda conectarse a Internet cuando intentan acceder a una parte de la aplicación que no está disponible sin conexión.

Comentarios

Los comentarios pueden ser agregados al manifiesto usando el símbolo **#** (uno por cada línea de comentario). Debido a que la lista de archivos es ordenada en categorías, puede parecer inútil el agregado de comentarios, pero son importantes, especialmente a la hora de realizar actualizaciones en el caché (descargar nuevas versiones de archivos). El archivo manifiesto no solo declara qué archivos serán incluidos en el caché, sino cuándo. Cada vez que los archivos de la aplicación son actualizados, no hay forma en la que el navegador pueda saberlo excepto a través del archivo manifiesto. Si los archivos actualizados son los mismos y ninguno fue agregado a la lista, el archivo manifiesto lucirá exactamente igual que antes, entonces el navegador no podrá reconocer la diferencia y seguirá usando los archivos viejos que ya se encuentran en el caché. Sin embargo, podemos forzar al navegador a descargar nuevamente los archivos de la aplicación indicando la existencia de una actualización por medio del agregado de comentarios. Normalmente, un solo comentario con la fecha de la última actualización (o cualquier otro dato) será suficiente, como es mostrado en el siguiente ejemplo:

```
CACHE MANIFEST

CACHE:
```

```
cache.html
cache.css
cache.js

NETWORK:
chat.html

FALLBACK:
noticias.html sinnoticias.html

# fecha 2011/08/10
```

Listado 16-3. *Nuevo comentario para informar sobre actualizaciones.*

Supongamos que agregamos más código a las funciones actuales del archivo **`cache.js`**. Los usuarios tendrán el archivo dentro del caché en sus ordenadores y los navegadores usarán esta vieja versión en lugar de la nueva. Cambiando la fecha al final del archivo manifiesto o agregando nuevos comentarios informaremos al navegador acerca de la actualización y todos los archivos necesarios para trabajar sin conexión serán nuevamente descargados, incluyendo la versión mejorada del archivo **`cache.js`**. Luego de que el caché es actualizado, el navegador ejecutará la aplicación usando los nuevos archivos en el ordenador del usuario.

Usando el archivo manifiesto

Luego de seleccionar todos los archivos necesarios para que la aplicación pueda funcionar sin conexión a Internet e incluir la lista completa de URLs apuntando a estos archivos, tenemos que cargar el manifiesto desde nuestros documentos. La API provee un nuevo atributo para el elemento **`<html>`** que indica la ubicación de este archivo:

```
<!DOCTYPE html>
<html lang="es" manifest="micache.manifest">
<head>
  <title>Offline API</title>
  <link rel="stylesheet" href="cache.css">
  <script src="cache.js"></script>
</head>
<body>
  <section id="cajadatos">
    Aplicación para trabajar sin conexión
  </section>
</body>
</html>
```

Listado 16-4. *Cargando el archivo manifiesto.*

El Listado 16-4 muestra un pequeño documento HTML que incluye el atributo **`manifest`** en el elemento **`<html>`**. El atributo **`manifest`** indica la ubicación del archivo manifiesto necesario para generar el caché de la aplicación. Como puede ver, nada cambia en el resto del documento: los archivos para estilos CSS y códigos Javascript son incluidos como siempre, independientemente del contenido del archivo manifiesto.

El manifiesto debe ser grabado con la extensión **`.manifest`** y el nombre que desee (en nuestro ejemplo, **`micache`**). Cada vez que el navegador encuentra el atributo **`manifest`** en un documento, intentará descargar el archivo manifiesto en primer lugar y luego todos los archivos listados en su interior. El atributo **`manifest`** debe ser incluido en cada documento HTML que tiene que ser parte del caché de la aplicación. El proceso es transparente para el usuario y puede ser controlado desde código Javascript usando la API, como veremos pronto.

Además de la extensión y la estructura interna del archivo manifiesto, existe otro requisito importante a considerar. El archivo manifiesto debe ser provisto por los servidores con el tipo MIME apropiado. Cada archivo posee un tipo MIME asociado para indicar el formato de su contenido. Por ejemplo, el tipo MIME para un archivo HTML es **`text/html`**. Un archivo manifiesto debe ser provisto usando el tipo **`text/cache-manifest`** o el navegador devolverá un error.

> **IMPORTANTE:** El tipo MIME **`text/cache-manifest`** no forma parte de la configuración por defecto de ningún servidor en este momento. Usted deberá agregarlo a su servidor manualmente. Cómo incluir este nuevo tipo de archivo depende de la clase de servidor con la que trabaje. Para algunas versiones de Apache, por

ejemplo, el agregado de la siguiente línea en el archivo **httpd.conf** será suficiente para comenzar a despachar estos archivos con el tipo MIME apropiado: **AddType text/cache-manifest .manifest**.

16.2 API Offline

El archivo manifiesto por sí mismo debería ser suficiente para generar un caché para sitios webs pequeños o códigos simples, pero aplicaciones complejas demandan mayor control. El archivo manifiesto declara los archivos necesarios para el caché, pero no puede informar sobre cuántos de estos archivos ya fueron descargados, o los errores encontrados en el proceso, o cuándo una actualización está lista para ser usada, entre otras importantes situaciones. Considerando estos posibles escenarios, la API provee el nuevo objeto ApplicationCache con métodos, propiedades y eventos para controlar todo el proceso.

Errores

Probablemente el evento más importante del objeto ApplicationCache es **`error`**. Si un error ocurre durante el proceso de lectura de archivos desde el servidor, por ejemplo, el caché necesario para que la aplicación trabaje fuera de línea no podrá ser creado o actualizado. Es extremadamente importante reconocer estas situaciones y actuar de acuerdo a las circunstancias.

Usando el documento HTML presentado en el Listado 16-4, vamos a construir una pequeña aplicación para entender cómo funciona este evento.

```
function iniciar(){
  var cache=window.applicationCache;
  cache.addEventListener('error', errores, false);
}
function errores(){
  alert('error');
}
window.addEventListener('load', iniciar, false);
```

Listado 16-5. *Controlando errores.*

El atributo **`applicationCache`** para el objeto Window usado en el código del Listado 16-5 retorna el objeto ApplicationCache para este documento. Luego de almacenar una referencia al objeto dentro de la variable **`cache`**, agregamos una escucha para el evento **`error`**. Esta escucha llamará a la función **`errores()`** cuando el evento es disparado y un mensaje de alerta será mostrado informando el error.

Hágalo usted mismo: Cree un archivo HTML con el código del Listado 16-4, un archivo Javascript llamado **`cache.js`** con el código del Listado 16-5, y un archivo manifiesto llamado **`micache.manifest`**. De acuerdo a lo que hemos estudiado, deberá incluir en el archivo manifiesto la lista de archivos necesarios para el caché dentro de la categoría **`CACHE`**. Para nuestro ejemplo, estos archivos son el archivo HTML, el archivo **`cache.js`** y el archivo **`cache.css`** (los estilos para este último archivo son presentados en el Listado 16-6). Suba estos archivos a su servidor y abra el documento HTML en su navegador. Si elimina el archivo manifiesto u olvida agregar el tipo MIME correspondiente para este archivo en su servidor, el evento **`error`** será disparado. También puede interrumpir el acceso a Internet o usar la opción Trabajar sin Conexión ofrecida en Firefox para ver la aplicación funcionando sin conexión y desde el nuevo caché.

IMPORTANTE: La implementación de API Offline se encuentra en un nivel experimental en este momento. Recomendamos probar los ejemplos de este capítulo en Firefox o Google Chrome. Firefox ofrece la opción de desactivar la conexión y trabajar fuera de línea (haga clic en la opción Trabajar sin Conexión en el menú Desarrollador Web). Además, Firefox es el único navegador que nos permitirá eliminar el caché para facilitar su estudio (vaya a Opciones/Avanzado/Red y seleccione el caché de su aplicación para eliminarlo). Por otro lado, Google Chrome ya ha implementado casi todos los eventos disponibles en esta API y nos permitirá experimentar con todas las posibilidades que ofrece.

El archivo CSS tiene solo que incluir estilos para el elemento **`<section>`** de nuestra plantilla. Puede crear los suyos o utilizar los siguientes:

```
#cajadatos{
  width: 500px;
  height: 300px;
  margin: 10px;
  padding: 10px;
```

```
  border: 1px solid #999999;
}
```

Listado 16-6. *Regla CSS para la* `cajadatos`.

Online y offline

Una nueva propiedad para el objeto Navigator fue incorporada. Se llama **`onLine`** e indica el actual estado de la conexión. Esta propiedad tiene dos eventos asociados que serán disparados cuando su valor cambie. La propiedad y los eventos no son parte del objeto ApplicationCache, pero son útiles para esta API.

online Este evento es disparado cuando el valor de la propiedad **`onLine`** cambia a **`true`** (verdadero).

offline Este evento es disparado cuando el valor de la propiedad **`onLine`** cambia a **`false`** (falso).

El siguiente es un ejemplo de cómo usarlos:

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  window.addEventListener('online', function(){ estado(1); },
                                                         false);
  window.addEventListener('offline', function(){ estado(2); },
                                                         false);
}
function estado(valor){
  switch(valor){
    case 1:
      cajadatos.innerHTML+='<br>Estamos Conectados';
      break;
    case 2:
      cajadatos.innerHTML+='<br>Estamos Desconectados';
      break;
  }
}
window.addEventListener('load', iniciar, false);
```

Listado 16-7. *Controlando el estado de la conexión.*

En el código del Listado 16-7, usamos funciones anónimas para manejar eventos y enviar un valor a la función **`estado()`** con la intención de mostrar el mensaje correspondiente en la **`cajadatos`**. Los eventos serán disparados cada vez que el valor de la propiedad **`onLine`** cambie.

IMPORTANTE: No hay garantía alguna de que la propiedad retorne siempre el valor adecuado. Escuchar a estos eventos en un ordenador de escritorio probablemente no producirá ningún efecto, incluso cuando el equipo sea completamente desconectado de Internet. Para probar este ejemplo, recomendamos usar la opción Trabajar sin Conexión ofrecida por Firefox.

Hágalo usted mismo: Use los mismos archivos HTML y CSS de ejemplos previos. Copie el código del Listado 16-7 en el archivo **`cache.js`**. Usando Firefox, elimine el caché de su aplicación y abra el documento HTML. Para probar el funcionamiento de los eventos, puede usar la opción Trabajar sin Conexión. Cada vez que active o desactive esta opción, la condición cambiará y un nuevo mensaje será automáticamente agregado a la **`cajadatos`**.

Procesando el caché

Crear o actualizar el caché puede tomar desde algunos segundos hasta varios minutos, dependiendo del tamaño de los archivos que deben ser descargados. El proceso pasa por diferentes estados de acuerdo con lo que el navegador es capaz de hacer en cada momento. En una actualización normal, por ejemplo, el navegador intentará primero leer el archivo manifiesto para buscar por posibles actualizaciones, descargará todos los archivos listados en el manifiesto (si la actualización existe) e informará cuando el proceso es finalizado. Para ofrecer información sobre cada paso en el proceso, la API ofrece la propiedad **`status`**. Esta propiedad puede tomar los

valores siguientes:

UNCACHED (valor 0) Este valor indica que ningún caché fue creado aún para la aplicación.

IDLE (valor 1) Este valor indica que el caché de la aplicación es el más nuevo disponible y no es obsoleto.

CHECKING (valor 2) Este valor indica que el navegador está comprobando la existencia de nuevas actualizaciones.

DOWNLOADING (valor 3) Este valor indica que los archivos para el caché están siendo descargados.

UPDATEREADY (valor 4) Este valor indica que el caché de la aplicación está disponible y no es obsoleto, pero no es el más nuevo (una actualización está lista para reemplazarlo).

OBSOLETE (valor 5) Este valor indica que el caché actual es obsoleto.

Podemos controlar el valor de la propiedad **`status`** en cualquier momento, pero es mejor usar los eventos provisto por el objeto ApplicationCache para controlar el estado del proceso y el caché. Los siguientes eventos son normalmente disparados en secuencia, y algunos de ellos están asociados a un estado específico del caché de la aplicación:

checking Este evento es disparado cuando el navegador está controlando por la existencia de actualizaciones.

noupdate Este evento es disparado cuando no fueron encontrados cambios en el archivo manifiesto.

downloading Este evento es disparado cuando el navegador encuentra una nueva actualización y comienza a descargar los archivos.

cached Este evento es disparado cuando el caché está listo.

updateready Este evento es disparado cuando el proceso de descarga para una actualización fue completado.

obsolete Este evento es disparado cuando el archivo manifiesto ya no está disponible y el caché está siendo eliminado.

El siguiente ejemplo nos ayudará a entender este proceso. Mediante este código, cada vez que un evento es disparado, un mensaje es agregado a la **`cajadatos`** con el valor del evento y el de la propiedad **`status`**.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');

  cache=window.applicationCache;
  cache.addEventListener('checking', function(){ mostrar(1); },
                                                          false);
  cache.addEventListener('downloading', function(){ mostrar(2); },
                                                          false);
  cache.addEventListener('cached', function(){ mostrar(3); },
                                                          false);
  cache.addEventListener('updateready', function(){ mostrar(4); },
                                                           false);
  cache.addEventListener('obsolete', function(){ mostrar(5); },
                                                          false);
}

function mostrar(valor){
  cajadatos.innerHTML+='<br>Estado: '+cache.status;
  cajadatos.innerHTML+=' | Evento: '+valor;
}
window.addEventListener('load', iniciar, false);
```

Listado 16-8. *Controlando la conexión.*

Usamos funciones anónimas para responder a los eventos y enviar un valor que nos permita identificarlos luego en la función **`mostrar()`**. Este valor y el valor de la propiedad **`status`** son mostrados en la pantalla cada vez que el navegador realiza un nuevo paso en la generación del caché.

Hágalo usted mismo: Use los archivos HTML y CSS de ejemplos previos. Copie el código del Listado 16-8 dentro del archivo **`cache.js`**. Suba la aplicación a su servidor y vea cómo los diferentes pasos del proceso son mostrados en la pantalla de acuerdo al estado del caché cada vez que el documento es cargado.

IMPORTANTE: Si el caché ya fue creado, es importante seguir diferentes pasos para limpiar el viejo caché y cargar la nueva versión. Modificar el archivo manifiesto agregando un comentario es uno de los pasos necesarios, pero no el único. Los navegadores mantienen una copia de los archivos en el ordenador por algunas horas antes de siquiera considerar comprobar si existen actualizaciones, por lo que no importa cuántos comentarios o archivos agregue al manifiesto, el navegador utilizará el viejo caché por un tiempo. Para probar estos ejemplos, le recomendamos cambiar los nombres de cada archivo. Por ejemplo, agregar un número al final del nombre (como en **cache2.js**) hará que el navegador considere ésta como una nueva aplicación y cree un nuevo caché. Esto, por supuesto, es solo útil por propósitos didácticos.

Progreso

Aplicaciones que incluyen imágenes, varios archivos de códigos, información para bases de datos, videos, o cualquier otro archivo de tamaño considerable pueden tomar un buen tiempo en ser descargadas. Para seguir este proceso, la API trabaja con el ya conocido evento **progress**. Este evento es el mismo que ya hemos estudiado en capítulos anteriores.

El evento **progress** solo es disparado mientras los archivos son descargados. En el siguiente ejemplo vamos a usar los eventos **noupdate** junto con **cached** y **updateready** analizados previamente para informar cuando el proceso es finalizado.

```
function iniciar(){
  cajadatos=document.getElementById('cajadatos');
  cajadatos.innerHTML='<progress value="0" max="100">0%</progress>';

  cache=window.applicationCache;
  cache.addEventListener('progress', progreso, false);
  cache.addEventListener('cached', mostrar, false);
  cache.addEventListener('updateready', mostrar, false);
  cache.addEventListener('noupdate', mostrar, false);
}
function progreso(e){
  if(e.lengthComputable){
    var por=parseInt(e.loaded/e.total*100);
    var barraprogreso=cajadatos.querySelector("progress");
    barraprogreso.value=por;
    barraprogreso.innerHTML=por+'%';
  }
}
function mostrar(){
  cajadatos.innerHTML='Terminado';
}
window.addEventListener('load', iniciar, false);
```

Listado 16-9. *Progreso de la descarga.*

Como siempre, el evento **progress** es disparado periódicamente para informar acerca del estado del proceso. En el código del Listado 16-9, cada vez que **progress** es disparado, la función **progreso()** es llamada y la situación es informada en pantalla usando un elemento **<progress>**.

Existen diferentes situaciones posibles al final del proceso. La aplicación podría haber sido almacenada en el caché por primera vez, en este caso el evento **cached** es disparado. También podría ser que el caché ya existe y una actualización se encuentra disponible, entonces cuando los archivos son finalmente descargados el evento que es disparado es **updateready**. Y una tercera posibilidad es que un caché ya estaba en uso y no se encontró ninguna actualización, en este caso el evento **noupdate** es el que será disparado. Escuchamos a cada uno de estos eventos y llamamos a la función **mostrar()** en cada caso para imprimir el mensaje “Terminado” en la pantalla, indicando de este modo la finalización del proceso.

Puede encontrar una explicación de la función **progreso()** en el Capítulo 13.

Hágalo usted mismo: Use los archivos HTML y CSS de ejemplos previos. Copie el código del Listado 16-8 dentro del archivo **cache.js**. Suba la aplicación a su servidor y cargue el documento principal. Deberá incluir un archivo de gran tamaño en el manifiesto para poder ver trabajando la barra de progreso (algunos navegadores establecen limitaciones sobre el tamaño del caché. Recomendamos probar este ejemplo con

archivos de no más de 5 megabytes). Por ejemplo, usando el video **trailer.ogg** introducido en el Capítulo 5, el archivo manifiesto ser vería como el siguiente:

```
CACHE MANIFEST
cache.html
cache.css
cache.js
trailer.ogg

# fecha 2011/06/27
```

IMPORTANTE: En nuestro ejemplo utilizamos **innerHTML** para agregar un nuevo elemento **<progress>** al documento. Esta no es una práctica recomendada pero es útil y conveniente por razones didácticas. Normalmente los elementos son agregados al documento usando el método Javascript **createElement()** junto con **appendChild()**.

Actualizando el caché

Hasta el momento hemos visto cómo crear un caché para nuestra aplicación, cómo informar al navegador cuando una actualización está disponible y cómo controlar el proceso cada vez que un usuario accede a la aplicación. Esto es útil pero no completamente transparente para el usuario. El caché y las actualizaciones del mismo son cargados tan pronto como el usuario ejecuta la aplicación, lo que puede producir demoras y mal funcionamiento. La API resuelve este problema incorporando nuevos métodos que nos permiten actualizar el caché mientras la aplicación está siendo utilizada:

update() Este método inicia una actualización del caché. Le indica al navegador que descargue primero el archivo manifiesto y luego continúe con el resto de los archivos si detecta algún cambio (los archivos para el caché fueron modificados).

swapCache() Este método activa el caché más reciente luego de una actualización. No ejecuta ningún código y tampoco reemplaza recursos, solo le indica al navegador que un nuevo caché se encuentra disponible para su lectura.

Para actualizar el caché, lo único que necesitamos hacer es llamar al método **update()**. Los eventos **updateready** y **noupdate** serán útiles para conocer el resultado del proceso. En el próximo ejemplo, vamos a usar un nuevo documento HTML con dos botones para solicitar la actualización y comprobar cuál es el código que se encuentra actualmente en el caché.

```
<!DOCTYPE html>
<html lang="es" manifest="micache.manifest">
<head>
  <title>Offline API</title>
  <link rel="stylesheet" href="cache.css">
  <script src="cache.js"></script>
</head>
<body>
  <section id="cajadatos">
    Aplicación para trabajar sin conexión
  </section>
  <button id="actualizar">Actualizar Caché</button>
  <button id="prueba">Verificar</button>
</body>
</html>
```

Listado 16-10. *Documento HTML para probar el método* **update()**.

El código Javascript implementa técnicas ya estudiadas. Solo hemos agregado dos nuevas funciones para responder a los botones de la plantilla:

```
function iniciar(){
```

```
    cajadatos=document.getElementById('cajadatos');
    var actualizar=document.getElementById('actualizar');
    actualizar.addEventListener('click', actualizarcache, false);
    var prueba=document.getElementById('prueba');
    prueba.addEventListener('click', probarcache, false);

    cache=window.applicationCache;
    cache.addEventListener('updateready', function(){ mostrar(1); },
                                                          false);
    cache.addEventListener('noupdate', function(){ mostrar(2); },
                                                         false);
  }
  function actualizarcache(){
    cache.update();
  }
  function probarcache(){
    cajadatos.innerHTML+='<br>cambiar este mensaje';
  }
  function mostrar(valor){
    switch(valor){
      case 1:
        cajadatos.innerHTML+='<br>Actualización Lista';
        break;
      case 2:
        cajadatos.innerHTML+='<br>Actualización No Disponible';
        break;
    }
  }
  window.addEventListener('load', iniciar, false);
```

Listado 16-11. *Actualizando el caché y comprobando la versión actual.*

En la función **iniciar()**, se agregó una escucha para el evento **click** a los dos botones de la plantilla. Un clic en el botón **actualizar** llamará a la función **actualizarcache()** y ejecutará el método **update()**. Y un clic sobre el botón **prueba** llamará a la función **probarcache()** y un texto será mostrado en la **cajadatos**. Este texto nos facilitará la creación de una nueva versión del código con la que podremos comprobar si el caché es actualizado o no.

> **Hágalo usted mismo:** Cree un nuevo documento HTML con el código del Listado 16-10. El manifiesto y el archivo CSS son los mismos de ejemplos previos (a menos que usted haya cambiado algunos nombre de archivos. En este caso deberá actualizar la lista de archivos dentro del manifiesto). Copie el código del Listado 16-11 dentro de un archivo llamado **cache.js**, y suba todo a su servidor. Abra el documento principal en su navegador y use los botones para probar la aplicación.

Una vez que el documento HTML es cargado, la ventana muestra nuestra típica **cajadatos** y dos botones debajo. Como explicamos anteriormente, el botón “Actualizar Caché” tiene el evento **click** asociado con la función **actualizarcache()**. Si el botón es presionado, el método **update()** es ejecutado dentro de esta función y el proceso de actualización comienza. El navegador descarga el archivo manifiesto y lo compara con el mismo archivo que ya se encuentra en el caché. Si detecta que este archivo fue modificado, todos los archivos listados en su interior son descargados nuevamente. Cuando el proceso finaliza, el evento **updateready** es disparado. Este evento llama a la función **mostrar()** con el valor **1**, correspondiente al mensaje “Actualización Lista”. Por otro lado, si el archivo manifiesto no cambió, ninguna actualización es detectada y el evento **noupdate** es disparado. Este evento también llama a la función **mostrar()** pero con el valor **2**, correspondiente al mensaje “Actualización No Disponible”.

Puede comprobar cómo trabaja este código modificando o agregando comentarios al archivo manifiesto. Cada vez que presione el botón para actualizar el caché luego de una modificación, el mensaje “Actualización Lista” aparecerá en la **cajadatos**. Puede también hacer pruebas cambiando el texto en la función **probarcache()** para detectar cuándo una actualización está siendo utilizada como el caché actual.

> **IMPORTANTE:** Esta vez no hay necesidad de eliminar el caché desde el panel de control del navegador para descargar una nueva versión. El método **update()** fuerza al navegador a descargar el archivo manifiesto y el resto de los archivos si una actualización es detectada. Sin embargo, el nuevo caché no estará disponible hasta que el usuario reinicie la aplicación.

16.3 Referencia rápida

API Offline es un grupo de técnicas que involucran un archivo especial llamado manifiesto y varios métodos, eventos y propiedades para crear un caché y poder ejecutar aplicaciones desde el ordenador del usuario. La API fue pensada para proveer acceso permanente a las aplicaciones y la posibilidad de trabajar mientras sin acceso a Internet.

Archivo manifiesto

El archivo manifiesto es un archivo de texto con la extensión **`.manifest`** conteniendo una lista de los archivos necesarios para construir el caché de la aplicación. Debe ser comenzado con la línea **`CACHE MANIFEST`** y su contenido puede estar organizado bajo las siguientes categorías:

CACHE Esta categoría incluye los archivos que deben ser descargados para formar parte del caché.

NETWORK Esta categoría incluye los archivos que solo pueden ser accedidos cuando se está conectado.

FALLBACK Esta categoría permite definir archivos en el caché que serán usados en lugar de archivos en el servidor cuando éstos no estén disponibles.

Propiedades

El objeto Navigator incluye una nueva propiedad para informar el estado de la conexión:

onLine Esta propiedad retorna un valor booleano que indica la condición de la conexión. Es **`false`** (falso) si el navegador está desconectado y **`true`** (verdadero) en caso contrario.

La API provee la propiedad **`status`** para informar sobre el estado del caché de la aplicación. Esta propiedad es parte del objeto ApplicationCache y puede tomar los siguientes valores:

UNCACHED (valor 0) Este valor indica que ningún caché fue creado aún para la aplicación.

IDLE (valor 1) Este valor indica que el caché de la aplicación es el más nuevo y no es obsoleto.

CHECKING (valor 2) Este valor indica que el navegador está buscando nuevas actualiza-ciones.

DOWNLOADING (valor 3) Este valor indica que los archivos para el caché están siendo descargados.

UPDATEREADY (valor 4) Este valor indica que el caché para la aplicación está disponible y no es obsoleto, pero no es el más nuevo; una actualización está lista para reemplazarlo.

OBSOLETE (valor 5) Este valor indica que el caché actual es obsoleto.

Eventos

Existen dos eventos para controlar el estado de la conexión:

online Este evento es disparado cuando el valor de la propiedad **`onLine`** cambia a **`true`** (verdadero).

offline Este evento es disparado cuando el valor de la propiedad **`onLine`** cambia a **`false`** (falso).

La API ofrece varios eventos, pertenecientes al objeto ApplicationCache, que informan acerca de la condición del caché:

checking Este evento es disparado cuando el navegador está comprobando si existen nuevas actualizaciones.

noupdate Este evento es disparado cuando no se encuentran nuevas actualizaciones.

downloading Este evento es disparado cuando el navegador ha encontrado una nueva actualización y comienza a descargar los archivos.

cached Este evento es disparado cuando el caché está listo para ser usado.

updateready Este evento es disparado cuando la descarga de una nueva actualización ha finalizado.

obsolete Este evento es disparado cuando el archivo manifiesto no se encuentra disponible y el caché está

siendo eliminado.

progress Este evento es disparado periódicamente durante el proceso de descarga de los archivos para el caché.

error Este evento es disparado si ocurre un error durante la creación o la actualización del caché.

Métodos

Dos métodos son incluidos en la API para solicitar una actualización del caché:

update() Este método inicia una actualización del caché. Indica al navegador que descargue el archivo manifiesto y el resto de los archivos si una actualización es detectada.

swapCache() Este método activa el caché más reciente luego de una actualización. No ejecuta los nuevos códigos y tampoco reemplaza recursos, solo indica al navegador que un nuevo caché está disponible para su uso.

Conclusiones

Trabajando para el mundo

Este es un libro sobre HTML5. Fue pensado como una guía para desarrolladores, diseñadores y programadores que quieran construir sitios web y aplicaciones utilizando las tecnologías más actuales. Pero nos encontramos en un proceso de transición en el cual las viejas tecnologías se fusionan con las nuevas, y los mercados no pueden seguirles el paso. Al mismo tiempo que millones y millones de copias de nuevos navegadores son descargadas de la web, millones y millones de personas no son ni siquiera conscientes de su existencia. El mercado está aún repleto de viejos ordenadores funcionando con Windows 98 e Internet Explorer 6, o incluso peor.

Crear para la web fue siempre un desafío, y se vuelve cada vez más complicado. Apesar de los prolongados y duros esfuerzos por construir e implementar estándares para Internet, ni siquiera los nuevos navegadores los soportan por completo. Y viejas versiones de navegadores que no siguen ninguna clase de estándar siguen presente, funcionando alrededor de todo el mundo, haciendo nuestras vidas imposible.

Por esta razón, es momento de ver qué podemos hacer para acercar HTML5 a la gente, cómo podemos crear e innovar en un mundo que parece indiferente. Llegó la hora de estudiar qué alternativas tenemos para trabajar con estas nuevas tecnologías y hacerlas disponibles para todos.

Las alternativas

Cuando se trata de alternativas, debemos decidir qué posición tomar. Podemos ser agresivos, atentos, inteligentes o trabajadores. Un desarrollador agresivo dirá: “Esta aplicación fue programada para trabajar en nuevos navegadores. Los nuevos navegadores son gratuitos. No sea perezoso y descargue una copia”. El desarrollador atento dirá: “Esta aplicación fue desarrollada aprovechando las nuevas tecnologías disponibles. Si desea disfrutar mi trabajo en todo su potencial, actualice su navegador. Mientras tanto, aquí tiene una versión antigua que puede utilizar en su lugar”. El desarrollador inteligente dirá: “Hacemos la tecnología de última generación disponible para todos. No necesita hacer nada, nosotros ya lo hicimos por usted”. Y finalmente, un trabajador dirá: “Esta es una versión de nuestra aplicación adaptada a su navegador, aquí puede acceder a otra con más herramientas, especial para nuevos navegadores, y aquí ofrecemos la versión experimental de nuestra súper evolucionada aplicación”.

Para un acercamiento más práctico y útil, estas son las opciones disponibles cuando el navegador del usuario no está preparado para HTML5:

Informar Recomiende al usuario actualizar su navegador si algunas características requeridas por su aplicación no están disponibles.

Adaptar Seleccione diferentes estilos y códigos para el documento de acuerdo con las características disponibles en el navegador del usuario.

Redireccionar Redireccione usuarios a un documento completamente diferente diseñado especialmente para viejos navegadores.

Emular Use librerías que hagan HTML5 disponible en viejos navegadores.

Modernizr

Sin importar cuál es la opción elegida, lo primero que debemos hacer es detectar si las características de HTML5 requeridas por su aplicación están disponibles en el navegador del usuario o no. Estas características son independientes y fáciles de identificar, pero las técnicas requeridas para hacerlo son tan diversas como las características mismas. Desarrolladores deben considerar diferentes navegadores y versiones, y depender de códigos que a menudo no son para nada confiables.

Una pequeña librería llamada Modernizr fue desarrollada con la intención de resolver este problema. Esta librería crea un objeto llamado Modernizr que ofrece propiedades para cada característica de HTML5. Estas propiedades retornan un valor booleano que será **`true`** (verdadero) o **`false`** (falso) dependiendo si la característica está disponible o no.

La librería es de código abierto, programada en Javascript y disponible gratuitamente en *www.modernizr.com*. Solo tiene que descargar el archivo Javascript e incluirlo en sus documentos, como en el siguiente ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Modernizr</title>
  <script src="modernizr.min.js"></script>
  <script src="modernizr.js"></script>
</head>
<body>
  <section id="cajadatos">
    contenido
  </section>
</body>
</html>
```

Listado C-1. *Incluyendo Modernizr en sus documentos.*

El archivo llamado **`modernizr.min.js`** es una copia del archivo de la libraría Modernizr descargado desde su sitio web. El segundo archivo incluido en el documento HTML en el Listado C-1 es nuestro propio código Javascript donde controlamos los valores de las propiedades provistas por la librería:

```
function iniciar(){
  var cajadatos=document.getElementById('cajadatos');
  if(Modernizr.boxshadow){
    cajadatos.innerHTML='Box Shadow está disponible';
  }else{
    cajadatos.innerHTML='Box Shadow no está disponible';
  }
}
window.addEventListener('load', iniciar, false);
```

Listado C-2. *Detectando la disponibilidad de estilos CSS para generar sombras.*

Como puede ver en el código del Listado C-2, podemos detectar cualquier característica de HTML5 usando solo un condicional **`if`** y la propiedad del objeto Modernizr correspondiente. Cada característica tiene su propia propiedad disponible.

> **IMPORTANTE:** Esta es solo una introducción breve a esta útil librería. Usando Modernizr, por ejemplo, podemos también seleccionar un grupo de estilos CSS desde archivos CSS sin usar Javascript. Modernizr ofrece clases especiales para implementar en nuestro archivo de estilos y así seleccionar las propiedades CSS apropiadas de acuerdo a cuales están disponibles en el navegador que abrió la aplicación. Para aprender más sobre esta librería, visite www.modernizr.com.

Librerías

Una vez que las características disponibles son detectadas, tenemos la opción de usar solo aquello que funciona en el navegador del usuario o recomendarle actualizar el software a una versión que incluya todas las características que nuestra aplicación necesita. Sin embargo, imagine que usted es un desarrollador obstinado o un loco programador que (al igual que sus usuarios y clientes) no se interesa por fabricantes de navegadores o versiones de programas o versiones beta o características no implementadas o lo que sea, usted solo quiere ofrecer la última tecnología disponible sin importar nada.

Bueno, aquí es donde librerías independientes pueden ayudar. Docenas de programadores en el mundo, probablemente más obstinados que nosotros, se encuentran desarrollando y mejorando librerías que imitan características de HTML5 en navegadores viejos, especialmente APIs de Javascript. Gracias a este esfuerzo, ya disponemos de los nuevos elementos HTML, selectores y estilos CSS3, y hasta complejas APIs como Canvas o Web Storage en cada navegador del mercado.

Por mayor información diríjase al siguiente enlace donde encontrará una lista actua-lizada de todas las librerías disponibles: *www.github.com/Modernizr/Modernizr/wiki/*
HTML5-Cross-browser-Polyfills

Google Chrome Frame

Google Chrome Frame será probablemente nuestro último recurso. Personalmente pienso que era una buena idea al comienzo, pero hoy día es mejor recomendar a nuestros usuarios actualizar sus navegadores antes que descargar un agregado como Google Chrome Frame.

Google Chrome Frame fue específicamente desarrollado para las viejas versiones de Internet Explorer. Fue diseñado para introducir todo el poder y las posibilidades del navegador Google Chrome dentro de navegadores que no están preparados para las nuevas tecnologías pero aún se encuentran instalados en los ordenadores de los usuarios y forman parte del mercado.

Como dije anteriormente, fue una buena idea. Insertando una simple etiqueta HTML en nuestros documentos, un mensaje era mostrado a los usuarios recomendando instalar Google Chrome Frame antes de ejecutar la aplicación. Luego de finalizado este simple paso, todas las características soportadas por Google Chrome estaban automáticamente disponibles en ese viejo navegador. Sin embargo, los usuarios no evitaban descargar software de la web. No discernir cuál es la diferencia entre esto y descargar una nueva versión de un navegador, especialmente ahora que hasta Internet Explorer tiene su propia versión gratuita compatible con HTML5. Estos días, con tantos navegadores listos para ejecutar aplicaciones HTML5, es mejor guiar a los usuarios hacia este nuevo software en lugar de enviarlos hacia oscuros y confusos agregados como Google Chrome Frame.

Para conocer más sobre Google Chrome Frame y cómo usarlo visite:

code.google.com/chrome/chromeframe/

Trabajando para la nube

En este nuevo mundo de dispositivos móviles y computación en la nube, no importa qué tan nuevo sea el navegador, siempre habrá algo más de qué preocuparnos. Probablemente el iPhone puede ser considerado el responsable de comenzarlo todo. Desde su aparición, varias cosas cambiaron en la web. El iPad lo siguió, y toda clase de imitaciones emergieron luego para satisfacer este nuevo mercado. Gracias a este cambio radical, el acceso móvil a Internet se volvió una práctica común. De repente estos nuevos dispositivos se volvieron un importante objetivo para sitios y aplicaciones web, y la diversidad de plataformas, pantallas e interfaces forzaron a los desarrolladores a adaptar sus productos a cada caso específico.

Estos días, independientemente de la clase de tecnología que usemos, nuestros sitios y aplicaciones web deben ser adaptados a cada posible plataforma. Esta es la única manera de mantener consistencia y hacer nuestro trabajo disponible para todos. Afortunadamente, HTML considera estas situaciones y ofrece el atributo **media** en el elemento **<link>** para seleccionar recursos externos de acuerdo a parámetros predeterminados:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Documento Principal</title>
  <link rel="stylesheet" href="ordenador.css" media="all and (min-
                                                    width: 769px)">
  <link rel="stylesheet" href="tablet.css" media="all and (min-
                             width: 321px) and (max-width: 768px)">
  <link rel="stylesheet" href="celular.css" media="all and (min-
                               width: 0px) and (max-width: 320px)">
</head>
<body>
…
</body>
</html>
```

Listado C-3: *Diferentes archivos CSS para diferentes dispositivos.*

Seleccionar cuáles estilos CSS serán aplicados al documento es una manera fácil de hacer este trabajo. De acuerdo al dispositivo o al tamaño de la pantalla, los archivos CSS son cargados y los estilos apropiados son aplicados. Elementos HTML pueden ser cambiados de tamaño y documentos enteros pueden ser adaptados y mostrados dentro de espacios y circunstancias específicas.

En el Listado C-3, tres archivos CSS diferentes son incorporados para tres situaciones distintas. Las situaciones son detectadas por los valores del atributo **media** en cada etiqueta **<link>**. Usando las propiedades **min-width** y **max-width**, podemos determinar el archivo CSS que será aplicado a este documento de acuerdo con la resolución de la pantalla en la cual el documento está siendo mostrado. Si el tamaño horizontal de la pantalla es de un valor entre 0 y 320 pixeles, el archivo **celular.css** es cargado. Para una resolución entre 321 y 768 pixeles, el archivo **tablet.css** es el incluido. Y finalmente, para una resolución mayor a 768 pixeles, el archivo **ordenador.css** es el que será usado.

En este ejemplo, contemplamos tres posibles escenarios: el documento es cargado en un celular pequeño, una Tablet PC o un ordenador de escritorio. Los valores usados son los que normalmente se encuentran en estos dispositivos.

Por supuesto, el proceso de adaptación no incluye solo estilos CSS. Las interfaces provistas por estos dispositivos son ligeramente diferentes entre sí debido principalmente a la eliminación de partes físicas, como el teclado y el ratón. Eventos comunes como **click** o **mouseover** han sido modificados o en algunos casos reemplazados por eventos táctiles. Y además, existe otra importante característica presente en dispositivos móviles que le permite al usuario cambiar la orientación de la pantalla y de este modo cambiar también el espacio disponible para el documento. Todos estos cambios entre un dispositivo y otro hacen imposible lograr una buena adaptación con solo agregar o modificar algunas reglas CSS. Javascript debe ser usado para adaptar los códigos o incluso detectar la situación y redireccionar a los usuarios hacia una versión del documento específica para el dispositivo con el que están accediendo a la aplicación.

IMPORTANTE: Este tema se encuentra fuera de los propósitos de este libro. Para mayor información, visite nuestro sitio web.

Recomendaciones finales

Siempre habrá desarrolladores que le dirán: “Si usa tecnologías que no se encuentran disponibles para el 5% de los navegadores en el mercado, perderá un 5% de usuarios”. Mi respuesta es: “Si usted tiene clientes que satisfacer, entonces adapte, redireccione o emule, pero si usted trabaja para usted mismo, informe”.

Siempre debe buscar el camino al éxito. Si trabaja para otros, debe ofrecer soluciones completamente funcionales, productos que los clientes de sus clientes puedan acceder, sin importar la elección que hayan hecho con respecto a ordenadores, navegadores o sistemas operativos. Pero si usted trabaja para usted mismo, para ser exitoso debe crear lo mejor de lo mejor, debe innovar, estar por delante de los demás, independientemente de lo que el 5% de los usuarios tenga instalado en sus ordenadores. Debe trabajar para el 20% que ya descargó la última versión de Firefox, el 15% que ya tiene Google Chrome instalado en su ordenador, el 10% que tiene Safari en su dispositivo móvil. Usted tiene millones de usuarios listos para convertirse en sus clientes. Mientras que el desarrollador le pregunta por qué se arriesgaría a perder un 5% de usuarios, yo le pregunto: ¿por qué dar la espalda a las oportunidades que se le presentan?

Usted nunca conquistará el 100% del mercado, y eso es un hecho. Usted no desarrolla sus sitios web en chino o portugués. Usted no está trabajando para el 100% del mercado, usted ya trabaja solo para una pequeña porción del mismo. ¿Por qué seguir limitándose? Desarrolle para el mercado que lo acerque al éxito. Desarrolle para la porción del mercado que crece continuamente y que le permitirá aplicar las nuevas tecnologías disponibles y dejar correr su imaginación. Del mismo modo que no se preocupa por mercados que hablan otros idiomas, tampoco debe preocuparse por la parte del mercado que aún utiliza viejas tecnologías. Informe. Muéstreles lo que se están perdiendo. Aproveche las últimas tecnologías disponibles, desarrolle para el futuro y tendrá el éxito asegurado.

Extras

Los códigos fuente para este libro se encuentran disponibles en ***www.minkbooks.com***.